VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Návrh datové vrstvy pro business aplikaci v n-vrstvé architektuře s využitím technologie Microsoft Entity Framework 4.0

Design of the n-tier business application data layer using Microsoft Entity Framework 4.0

2011 Bc. Václav Bašniar

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 5. května 2011 .....

Rád bych na tomto místě poděkoval Ing. Tomáši Kocyanovi a Ing. Petru Foltýnkovi.

Abstrakt

Předmětem této diplomové práce je především návrh, implementace a testování datové vrstvy n-vrstvé aplikace. V práci jsou kromě samotné datové vrstvy popsány také technologie a techniky, které byly při její implementaci použity. Konkrétně se jedná o n-vrstvé architektury, objektově relační mapování, návrhový vzor Repository a návrhový vzor Unit of Work. Dále pak jsou to technologie pro práci s daty spojené s technologií .NET Framework. Především technologie dotazování LINQ a rámec pro objektově relační mapování Entity Framework 4, který je v práci podrobněji rozebrán.

Klíčová slova: objektově relační mapování, .NET Framework, ADO.NET, Entity Framework 4, LINQ, LINQ to SQL, NHibernate, návrhový vzor Unit of Work, návrhový vzor Repository, n-vrstvé architektury, POCO objekty

Abstract

The master thesis main subject is the n-tier architecture data tier design, implementation and testing. The thesis also describes technologies and techniques used for the data tier implementation. Which means the n-tier architectures, the object-relational mapping, the Repository pattern and the Unit of Work pattern. The technologies include .NET Framework's data manipulation technologies. Especially quering technology LINQ and object-relational framework Entity Framework 4, which is in the thesis analyzed in detail.

Keywords: object-relational mapping, .NET Framework, ADO.NET, Entity Framework 4, LINQ, LINQ to SQL, NHibernate, the Unit of Work pattern, the Repository pattern, n-tier architectures, POCO

Seznam použitých zkratek a symbolů

AA	–	Academic Aliance
ADO	–	ActiveX Data Object
API	–	Application Programming Interface
ASP	–	Active Server Pages
CRUD	–	Create Update Delete
CSDL	–	Conceptual Schema Definition Language
DTO	–	Data Transfer Object
EDM	–	Entity Data Model
EF	–	Entity Framework
HQL	–	Hibernate Query Language
ID	–	Identification
LINQ	–	Language-Integrated Query
MS	–	Microsoft
MSDN	–	MicroSoft Developers Network
MSL	–	Mapping Specification Language
ODBC	–	Open Database Connectivity
OLE-DB	–	Object Linking and Embedding - Database
ORM	–	Object-Relational Mapping
PHP	–	PHP: Hypertext Preprocessor
POCO	–	Plain Old CLR Objects
SP	–	Service Pack
SQL	–	Structured Query Language
SSDL	–	Store Schema Definition Language
T4	–	Text Templating Transformation Toolkit
VM	–	Virtual Machine
WCF	–	Windows Communication Foundation
XML	–	Extensible Markup Language

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Vývoj informačních systémů je náročným procesem. Od vzniku prvních informačních systémů vznikají a vyvíjejí se různé přístupy pro práci s bázemi dat, pro jejich ukládání, zpracování a prezentaci. K těmto systémům patří i enterprise[1] a business aplikace[2]. Mezi mezníky ve vývoji softwarových aplikací (a tedy i informačních systémů) patří používání návrhových vzorů a především přístup objektově orientovaného programování. U informačních systémů se navíc například vyvinuly n-vrstvé architektury, které strukturují aplikace do více vrstev a umožňují mimo jiné snazší správu zdrojového kódu aplikace. V této práci budou použity implementace využívající návrhových vzorů, n-vrstvých architektur a samozřejmě i principy objektově orientovaného programování.

V rámci této diplomové práce byla vyvinuta jednoduchá testovací business aplikace, která byla navržena pro co nejobjektivnější a nejpřehlednější testování. Aplikace byla vyvíjena v sadě MS Visual Studio 2010 Professional. Jako programovací jazyk byl použit jazyk C#, který je zároveň použit ve všech ukázkách v této práci. Následují informace o obsahu jednotlivých kapitol.

V kapitole N-vrstvé architektury je popsána historie, struktura a účel n-vrstvých architektur. Kapitola Objektově relační mapování rozebírá princip objektově relačního mapování a jeho použití. Dále pak přibližuje historii aplikace objektově relačního mapování v technologii .NET Framework. Kapitola Technologie LINQ popisuje možnosti technologie LINQ a její současné aplikace. V kapitole Návrhové vzory Repository a Unit of Work jsou tyto návrhové vzory popsány a jsou zde uvedeny důvody jejich společné aplikace. Dále jsou zde informace o tom, co to návrhové vzory jsou a informace o jejich struktuře a použití. Kapitola Technologie Entity Framework 4 podrobněji rozebírá samotnou technologii Entity Framework ve verzi 4. V této kapitole jsou navíc vypsány rozdíly s vybranými rámci objektově relačního mapování. Následují kapitoly zabývající se vytvořenou datovou vrstvou, jejím návrhem, možnostmi a popisem práce s ní. V kapitole Testování jsou uvedeny informace k testům různých implementací datových vrstev, jsou zde vyhodnoceny a porovnávány jejich výsledky.

Při zpracovávání této diplomové práce byl veškerý software společnosti Microsoft použit výhradně pod licencí MSDN AA. Jako hlavní zdroj informací byla použita kniha [12] a web [7].

[1] specializovaný software/informační systém určený pro použití ve firmách

[2] informační systém pro podporu business procesů

2 N-vrstvé architektury

N-vrstvé architektury informačních systémů mají zpravidla dvě nebo tři vrstvy, jedná se tedy o dvou či třívrstvé architektury. Jednotlivé vrstvy n-vrstvých architektur lze vyvíjet a upravovat odděleně a také je v případě potřeby vyměnit.

Dvouvrstvé architektury se skládají z datové a klientské (nebo také prezentační) vrstvy. Klientská vrstva slouží ke komunikaci s uživatelem. Datovou vrstvou je zde myšlena vrstva se zdrojem dat. Aplikační logika je u této architektury umístěna především v klientské vrstvě. Část aplikační logiky může být umístěna i v datové vrstvě, pokud to umožňuje zdroj dat. Rostoucí složitost aplikací vedla k vytvoření vlastní vrstvy pro aplikační logiku – aplikační vrstvu. Dvouvrstvé architektury jsou dnes již zastaralé a nepoužívají se.

Třívrstvé architektury vznikly přirozeným vývojem z dvouvrstvých architektur přidáním aplikační vrstvy. Ta je dále složena z business vrstvy a vrstvy pro přístup k datům (v této práci označovaná jako datová vrstva). Business vrstva zpracovává data získaná z datové vrstvy a interpretuje je vrstvě prezentační, tento proces je obousměrný. Přechod na třívrstvé architektury byl zapříčiněn především vzrůstajícími nároky na klientskou vrstvu, kde jednotliví klienti plnili roli tzv. tlustého klienta (téměř všechna logika byla umístěna na straně klienta). Tento krok vedl ke vzniku tzv. tenkých klientů (klienti obsahují minimum logiky a jsou určení především k prezentaci systému uživatelům). Tenkými klienty tak mohou být například i webové prohlížeče. Tohoto přístupu lze použít zvláště při aplikaci informačních systémů v rámci počítačových sítí (distribuované systémy), kdy mohou být pro datovou a aplikační vrstvu použity specializované výkonné servery a jako klienty klientské vrstvy použít právě webové prohlížeče.[15, 13]

3 Objektově relační mapování

3.1 Co je to objektově relační mapování

S daty uloženými v relační databázi mohou aplikace pracovat pomocí poskytovatelů dat (např. ODBC, OLE-DB). Tito poskytovatelé umožňují aplikacím pracovat s databázemi pomocí standardních SQL příkazů select, insert, update a delete pro vybírání, vkládání, upravování a mazání záznamů v tabulkách specifikované databáze. Manipulovat s těmito poskytovateli lze pomocí specifických bázových tříd daného programovacího jazyka.

Tyto třídy ale získávají data z databáze pouze jako plochá data a tak je třeba tato data vždy přetransformovat do požadované podoby, stejně tak je nutné data používaná při dalších operacích mezi aplikací a databází kontrolovat. Všechny tyto úkony automatizuje technika objektově relačního mapování (ORM), kdy je struktura relační databáze převedena na její objektový ekvivalent a jednotlivé prvky databáze a jejího modelu jsou na sebe namapovány. Tabulky pak mohou být z pohledu aplikace reprezentovány třídami a vazby a atributy vlastnostmi těchto tříd. Samotná data z tabulek jsou pak uložena ve vlastnostech instancí těchto tříd. Kromě samotného přetransformovávání dat při čtení a zapisování, usnadňuje tato technika programátorům práci také tím, že umožňuje využití výhod objektově orientovaného programování. Daní za toto pohodlí je ovšem zvýšená časová a paměťová složitost.

Objektově relační mapování lze implementovat „vlastními silami“ či pomocí ORM rámců. ORM rámce jsou nástroje, které umožňují snazší vytvoření objektově relačního mapování nad vytvořenou datovou strukturou zdroje dat. Některé umožňují automatizované generování zdrojového kódu tříd z datové struktury, či naopak vygenerování této struktury ze tříd, které tuto strukturu definují.

3.2 Objektově relační mapování v technologii .NET Framework

Pro .NET Framework existuje velké množství ORM řešení vytvořených především třetími stranami. Od verze 3.5 obsahuje technologie .NET Framework implementaci ORM rámce od společnosti Microsoft – LINQ to SQL, která kromě běžného objektově relačního mapování zahrnuje především technologii dotazování integrovaného do jazyka, tedy Language-Integrated Query (LINQ), více o technologii LINQ lze nalézt v kapitole 4. Souběžně s ORM rámcem LINQ to SQL začala být vyvíjena i technologie Entity Framework, která se poprvé objevila v technologii .NET Framework verze 3.5 SP1.

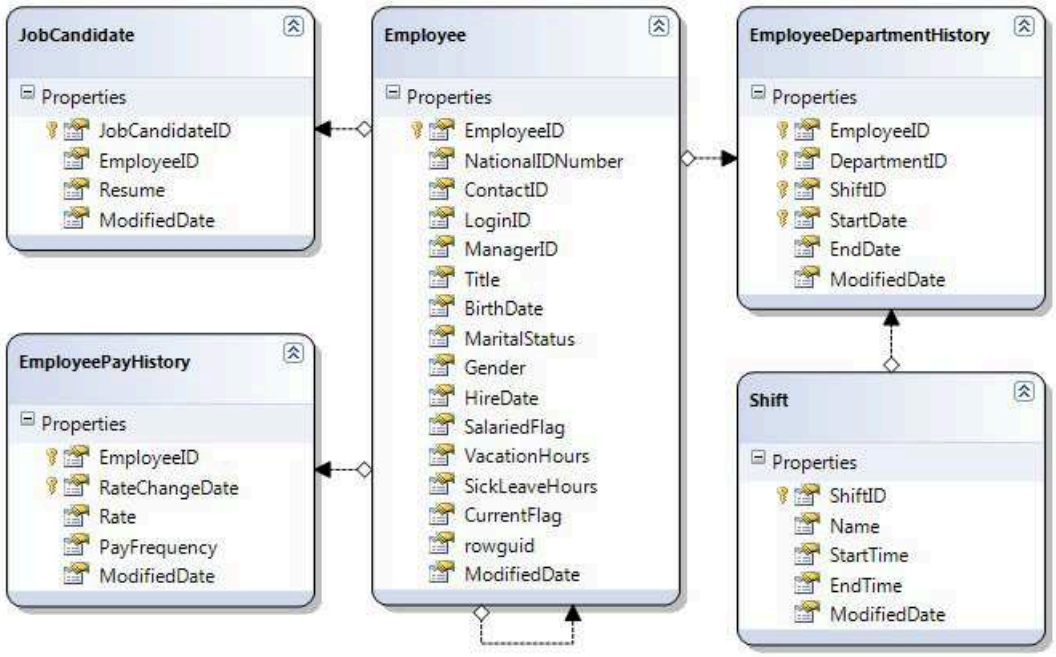
3.2.1 ADO.NET

V technologii .NET Framework jsou jako bázové třídy pro základní manipulaci s poskytovateli dat používány třídy jmenného prostoru ADO.NET. V rámci technologie .NET Framework se přístupu, kdy se přistupuje přímo k datům do databáze říká připojené prostředí. Jak bylo naznačeno výše, je práce s databází pomocí této techniky zdlouhavá a málo efektivní, lze ale pomocí ní naimplementovat vlastní ORM řešení.

S daty databáze lze v ADO.NET pracovat také pomocí odpojeného prostředí, kde jsou data z databáze nahrána do objektu DataSet, který je uložen v paměti a změny v databázi jsou prováděny až po ukončení práce s tímto objektem.

3.2.2 LINQ to SQL

LINQ to SQL je implementací ORM rámce od společnosti Microsoft pro práci s relačními databázemi. Jedná se o běžný ORM rámec, který převádí plochá data z datového zdroje, v tomto případě relační databáze, na objekty. Mapování lze implementovat manuálně přímo v kódu aplikace, nebo automatizovaně pomocí vizuálního návrháře. Například sada Visual Studio poskytuje vizuální návrhář pro práci s LINQ to SQL a databází. Objekty jsou na relační databázi mapovány jedna ku jedné, tedy jedna tabulka představuje jednu třídu a jeden atribut tabulky představuje jednu vlastnost třídy. Analogicky pak jeden záznam v tabulce představuje jeden objekt v paměti. Vazby jsou realizovány pomocí vlastností, jednak uloženým cizím klíčem, jednak uloženou referencí na objekt (v případě vazby 1:n kolekcí referencí na straně 1). Nutno podotknout, že LINQ to SQL je určen pouze pro prácí se databázovým systémem MS SQL Server. LINQ to SQL patří do skupiny rámců LINQ to ..., které implementují technologii LINQ pro různé typy datových zdrojů. Hlavní výhodou tohoto rámce je využití technologie LINQ.



Obrázek 1: Schéma části databáze AdventureWorks ve vizuálním návrháři pro práci s LINQ to SQL a databází v sadě Visual Studio 2010

2.2.2.1 Třída DataContext

DataContext je třídou LINQ to SQL, která udržuje všechny objekty mapované na databázi. Jedná se o tzv. „lightweight“ třídu, tzn. že lze snadno a bez velkých nároků vytvořit její instanci a udržovat ji. DataContext nahrává požadovaná data z databáze do objektů, hlídá nekonzistence mezi objekty a umožňuje promítnutí změn nad objekty zpět do databáze. DataContext umožňuje dotazování nad mapovanými objekty pomocí LINQ dotazů a také přímé dotazování na zdroj dat pomocí SQL dotazů. LINQ dotazy nad objekty překládá na ekvivalentní SQL dotazy, ty jsou odeslány na zdroj dat a výsledky jsou převedeny na odpovídající objekty.

DataContext lze použít tak jak je, ale doporučuje se pro specifická data naimplementovat vlastní třídu, která bude třídu DataContext rozšiřovat. Třída rozšiřující DataContext pak umožňuje snazší práci s datovou strukturou.

3.2.3 Entity Framework

Další ORM rámec od společnosti Microsoft. Entity Framework je v současné době upřednostňovanou technologií pro přístup k datům v relačních databázích a postupně by měla nahradit odpojené prostředí ADO.NET (DataSet) i LINQ to SQL. Hlavními rozdíly oproti LINQ to SQL jsou: možnost použití jiných databázových systému než Microsoft SQL Server a možnost překrytí datového modelu doménovým modelem. Doménový model lze s výhodou použít, pokud více aplikací používá stejnou databázi a zároveň vyžadují jinou datovou strukturu. Rámec Entity Framework bude v této práci podrobněji rozebrán dále.

3.2.4 NHibernate a další

Jak bylo zmíněno výše, existuje velké množství komerčních i open-source ORM rámců implementovaných třetí stranou. Mezi nejznámější z nich patří pravděpodobně ORM rámec NHibernate. Tento rámec je implementací ORM rámce Hibernate původně určeného pro jazyk JAVA.

Mezi další ORM rámce třetích stran pro .NET Framework patří například EntitySpaces, Genome, LLBLGen Pro a Opf3. Některé z těchto rámců již mají naimplementovánu podporu LINQ. [16]

4 Technologie LINQ

Technologie Language-Integrated Query (LINQ) je rozšířením skupiny jazyků technologie .NET Framework o možnosti dotazování na zdroje dat přímo v kódu aplikace. Jsou to jednak dotazy podobné běžným SQL příkazům a jednak rozšiřující metody kolekcí implementující rozhraní IEnumerable využívající pro definici dotazů delegáty. Tyto dotazy a metody vracejí výsledná data jako kolekci objektů.

```
var anonymousType = from employee in context.Employees
                    where employee.BirthDate.CompareTo(new DateTime(1980, 8, 8)) < 0
                    orderby employee.BirthDate
                    select new {
                            ID = employee.EmployeeID,
                            Age = DateTime.Today.Year − employee.BirthDate.Year
                     };
```

Výpis 1: Ukázka použití technologie LINQ

V ukázce 1 je do proměnné anonymousType typu var[3] vložen LINQ dotaz, který v kolekci uložené v proměnné context.Employees vyhledá zaměstnance, kteří se narodili před datem 8.8.1980 a setřídí je podle data narození. Technologie LINQ vytvoří pro každý takto získaný záznam objekt anonymní třídy s vlastnostmi ID a Age, do které uloží získané vybrané údaje.

Mezi hlavní výhody použití technologie LINQ patří: kontrola syntaxe během kompilace a využití technologie IntelliSense při vývoji v sadě MS Visual Studio. Tyto výhody umožňují snazší odladění chyb a urychlení práce během vývoje aplikací. Další výhody pak vychází z automatizovaného přetypovávání ve stylu ORM rámců, tj. převod mezi datovým typem datového zdroje na datový typ použitého programovacího jazyka.

Samotná technologie LINQ je tedy více než klasický ORM rámec. Je navržena tak, aby ji bylo možno použít pro téměř jakýkoli zdroj dat. [1]

4.1 LINQ to ...

LINQ lze použít především pro data relační (SQL) a pro data hierarchická (XML), kdy jsou dotazy LINQ ve výsledku převedeny na svůj ekvivalent v dotazovacím jazyce pro daná data (SQL, XPath). LINQ následně získaná data ze zdroje přetransformuje na specifikované objekty. LINQ lze použít i pro kolekce implementující rozhraní IEnumerable a pro dotazování nad datasety.

Právě použití LINQ pro dotazování nad kolekcemi IEnumerable, společně s možností implementace vlastních dotazovacích operátorů a přetěžování těch stávajících a možností definice derivačního stromu pro jednotlivé výrazy, umožňuje komukoli implementovat vlastní řešení pro vlastní zdroje dat.

[3] implicitní typ, silně typovaný; kompilátor rozhodne, o jaký typ se jedná, umožňuje použití anonymních typů.

4.2 Lambda výrazy

Jak bylo řečeno výše pro dotazování na datové zdroje lze použít metody kolekcí implementující rozhraní IEnumerable. Tyto metody slouží k výběru, třízení a filtrovaní dat. Metody jsou pojmenovány stejně jako operátory SQL dotazů a stejně i fungují. Jako parametr lze těmto metodám předávat delegáty metod, nebo delegáty typu Func<T, TResult> (kde T je datový typ hodnoty předávané anonymní metodě jako parametr a TResult je datový typ návratové hodnoty) odkazující se na anonymní třídu (Ukázka 2). Jako podmínka je pak použita samotná logika předávaných metod.

```
//zápis pomocí anonymních metod
vyber = delegate(Employee e) {
    return e.HireDate.CompareTo(new DateTime(2003, 8, 8)) < 0;
};

razeni = delegate(Employee e) {
    return e.BirthDate;
};

vyberAtributu = delegate(Employee e) {
    return DateTime.Today.Year - e.BirthDate.Year;
};

//LINQ dotaz vytvořený pomocí rozšiřujících tříd
IEnumerable<int> ages = context.Employees
                            .Where(omezeni)
                            .OrderBy(razeni)
                            .Select(vyberAtributu);
```

Výpis 2: Anonymní třídy v technologii LINQ

Lambda výrazy usnadňují práci při vytváření anonymních metod. Jedná se vlastně o jejich zjednodušený zápis. Lambda výrazy jsou zapisovány ve tvaru: *(vstupní parametry)* => *výraz* (Ukázka 3).

```
//zápis pomocí lambda výrazů
Func<Employee, bool> vyber =
                        e => e.HireDate.CompareTo(new DateTime(2003, 8, 8)) < 0;

Func<Employee, DateTime> razeni = e => e.BirthDate;

Func<Employee, int> vyberAtributu = e => DateTime.Today.Year - e.BirthDate.Year;
```

Výpis 3: Lambda výrazy v technologii LINQ

Jestliže je delegát typu Func<T, TResult> použit společně s generickým typem Expression<T>, kterému je předán jako parametr (Expression< Func<T, TResult>>), pak kompilátor s lambda výrazy zachází jako s derivačním stromem (viz. 4.4) a ne jako s anonymní třídou.

4.3 Zpožděné vyhodnocování (Deffered Execution/Lazy Loading)

K provedení LINQ dotazu nedojde okamžitě po asociaci dotazu s proměnnou typu IEnumerable, IQueryable případně var, ale až po zavolání metody ToArray() nebo ToList() získaného objektu, případně enumerací tohoto objektu, například pomocí příkazu foreach (Ukázka 4). To ve výsledku vede k tomu, že pokud dojde ke změně dat datového zdroje, promítne se tato změna i do objektu asociovaného s LINQ dotazem.

```
//Asociuje se dotaz
var employees = context.Find(e => e.HireDate.CompareTo(new DateTime(2003, 8, 8)) < 0);

//Vykoná se dotaz a získaná data se uloží do pole.
//Změna v objektech v poli employeesArray se neprojeví na objektech ve zdroji.
Employee[] employeesArray = employees.ToArray();
```

Výpis 4: Ukázka zpožděného vyhodnocování v technologii LINQ

Ke zpožděnému vyhodnocování dochází také při získávání objektu, nebo objektů, pomoci vlastností objektu, který je s tímto objektem, objekty, v relaci (Ukázka 5). Tato vlastnost se vyskytuje u objektově relačních rámců LINQ to SQL a Enitity Framework 4, kde ji lze vypnout pomocí příslušné vlastnosti objektu získávajícího data ze zdroje dat. Potom nejsou objekty do vlastností nahrávány.

```
//Kolekce objeků EmployeePayHistory, které jsou v asociaci s prvním objektem v kolekci employees
    .
ICollection<EmployeePayHistory> employeePayHistory = employees.First().EmployeePayHistory;
```

Výpis 5: Zpožděné vyhodnocování při získávání asociovaných objektů

Tato funkčnost může vést k nežádoucímu nárůstu počtu objektů v paměti.

4.4 Derivační stromy (Expression Trees)

V technologii LINQ lze specifikovat derivační stromy výrazů a to přímo pomocí API[4]. V případě lambda výrazů jsou derivační stromy sestavovány automaticky.

Při vytváření derivačních stromů pomocí API se používá třída Expression, která obsahuje třídy potřebné pro vytvoření uzlů derivačního stromu (Ukázka 6).

```
//uložení lambda výrazu do objektu typu Expression<T>
Expression<Func<DateTime, bool>> expr = e => e.CompareTo(new DateTime(2003, 8, 8)) < 0;

//rozložení výrazu
BinaryExpression teloVyrazu = (BinaryExpression)expr.Body;
ParameterExpression levaStranaVyrazu = (ParameterExpression)teloVyrazu.Left;
ConstantExpression pravaStranaVyrazu = (ConstantExpression)teloVyrazu.Right;
```

Výpis 6: Ukázka rozložení lambda výrazu

[4]aplikační rozhraní, rozhraní umožňující práci například s moduly

4.5 Rozhraní IQueryable<T>

Jedná se o rozhraní, které umožňuje provádět dotazování na zdroje dat. Používá se při implementaci poskytovatelů technologie LINQ pro vlastní zdroje dat. Rozšiřuje rozhraní IEnumerable<T>, to umožňuje získaná data reprezentovat jako kolekce objektů.

5 Návrhové vzory Repository a Unit of Work

5.1 Co jsou to návrhové vzory

Knihy a články o návrhových vzorech (např.:[14, 10]) se v těchto místech odkazují na výrok Christophera Alexandera[8]: „Každý návrhový vzor popisuje nějaký problém, který se v našem prostředí objevuje pořád dokola a následně popisuje jádro řešení tohoto problému takovým způsobem, že toto řešení lze použít opakovaně milionkrát, aniž by bylo dvakrát vytvořeno stejným způsobem“, který pravděpodobně nejlépe vystihuje podstatu návrhových vzorů a autoři jej používají místo definice. Christopher Alexander je rakouský architekt a v jeho případě se jednalo o stavby.

Návrhové vzory nevznikají „na zelené louce“, ale z praxe osvědčených postupů. Návrhové vzory lze teda spíše nalézt v kódech aplikací, než je vymýšlet z hlavy.

Každý návrhový vzor má své jméno, popis problému, který řeší, samotné řešení problému a následky, které aplikace daného návrhového vzoru přináší [10].

Návrhové vzory se objevují ve všech odvětvích softwarového vývoje. Tato práce se bude zabývat především návrhovými vzory Repository a Unit of Work určenými pro objektově relační mapování především v enterprise aplikacích.

5.2 Návrhový vzor Repository

Návrhový vzor Repository je v n-vrstvých architekturách implementován v datové vrstvě. Slouží k zakrytí přístupového kódu k datům datové vrstvy. To vede ke zpřehlednění a zjednodušení kódu na vyšších vrstvách n-vrstvé aplikace a hlavně usnadňuje výměnu datové vrstvy. Jedná se o návrhový vzor patřící do skupiny vzorů mapování pomocí metadat.[14]

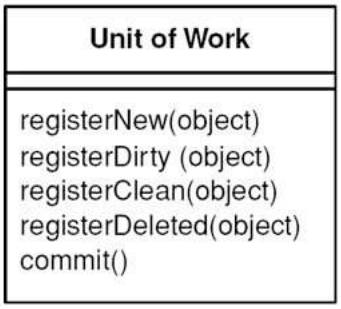
5.3 Návrhový vzor Unit of Work

Návrhový vzor Unit of Work slouží k udržování změn provedených nad objekty získanými z databáze a jejich následnému uložení do databáze. Unit of Work také řeší problémy s konzistencí dat v paměti a v databázi. Jedná se o návrhový vzor chování.[14] Tento návrhový vzor již bývá u ORM rámců implementován, u rámců LINQ to SQL a Entity Framework se jedná o třídy DataContext, respektive ObjectContext.

Třídní diagram na obrázku 2 zobrazuje metody, které by měl návrhový vzor Unit of Work obsahovat. Metody registerNew() a registerDeleted() slouží ke vložení nového respektive odstranění existujícího objektu. Metoda registerDirty() slouží k označení objektů, ve kterých byla v paměti provedena nějaká změna. Pomocí metody registerClean() jsou označovány objekty především při načítání, metoda zjistí, zdali není objekt již načten. Metoda commit() promítá změny na zdroj dat.

5.4 Proč návrhové vzory Repository a Unit of Work

Návrhové vzory Repository a Unit of Work je nejlépe použít společně. Návrhový vzor Repository funguje jako silně typovaná kolekce objektů získaných z databáze. K získá-



Obrázek 2: Třídní diagram implementace návrhového vzoru Unit of Work

vání objektů využívá návrhového vzoru Unit of Work, který navíc udržuje připojení na databázi, eviduje změny v objektech a promítá tyto změny do databáze. Obecně platí, že více instancí návrhového vzoru Repository využívá právě jednu instanci návrhového vzoru Unit of Work. Návrhový vzor Repository se implementuje zvlášť pro každý objekt/tabulku v databázi, zatímco implementaci návrhového vzoru Unit of Work lze použít pro jakoukoli databázi.

6 Technologie Entity Framework 4

6.1 Co je to Entity Framework

Jak již bylo řečeno výše jedná o ORM rámec vyvinutý společností Microsoft pro použití v technologii .NET Framework. Umožňuje pracovat s různými relačními databázovými systémy. Je ovšem potřeba mít odpovídající poskytovatele dat ADO.NET. Technologie Entity Framework dokáže úplně zakrýt model relační databáze konceptuálním (doménovým) modelem, to ve výsledku vede k tomu, že veškerá data jsou aplikaci interpretována jakoby byl doménový model samotným datovým modelem relační databáze. Technologie Entity Framework provádí potřebné transformace dotazů na doménový model na dotazy vyžadované datovým zdrojem. Výsledná data získaná z datového zdroje jsou přetransformována na požadované objekty do kolekcí implementující rozhraní IEnumerable<T>. Dotazování lze provádět pomocí pomocí jazyka Entity SQL nebo pomocí implementace technologie LINQ pro Entity Framework – LINQ to Entities.

6.2 Architektura technologie Entity Framework 4

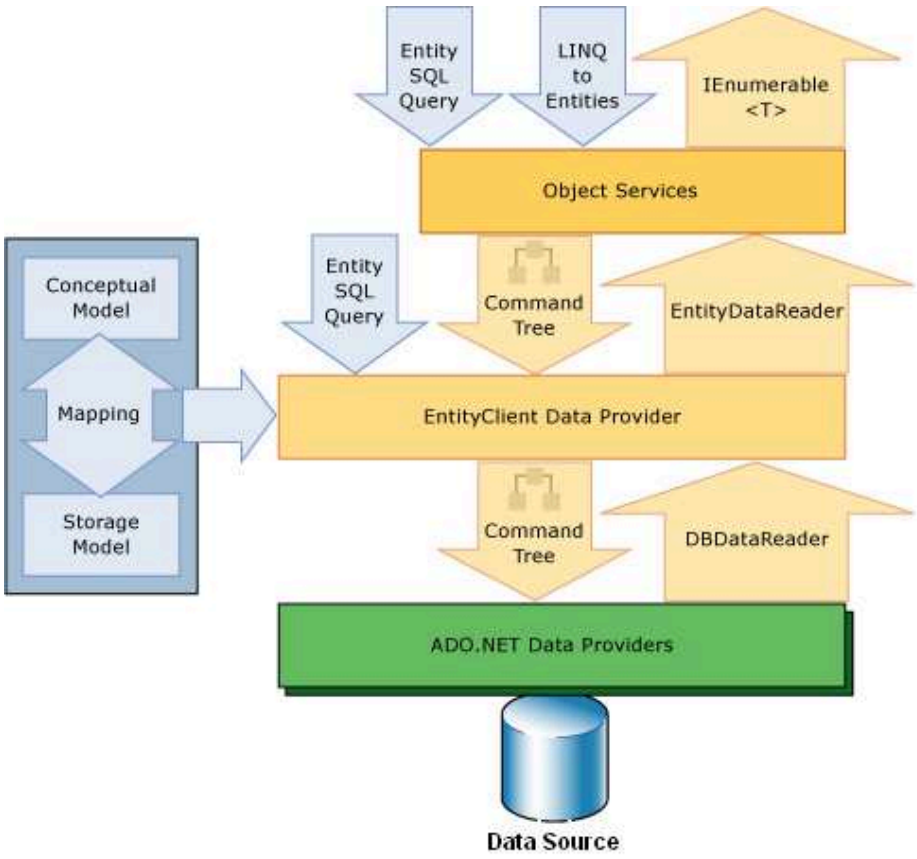
Architektura technologie Entity Framework 4 je složena ze tří nad sebou položených vrstev: vrstvy ADO.NET Data Providers, vrstvy EntityClient Data Provider a vrstvy Object Services (Obrázek 3). V následujících odstavcích je uveden popis jednotlivých vrstev.

Nejníže položenou vrstvou architektury technologie Entity Framework 4 jsou poskytovatelé dat ADO.NET (ADO.NET Data Providers), kteří komunikují přímo s datovým zdrojem. Z vyšší vrstvy (EntityClient Data Provider) přijímají stromy příkazů [5], z nichž skládají dotazy specifické pro daný zdroj dat, které následně tomuto datovému zdroji posílají k vyhodnocení. Získaná data vracejí do vrstvy EntityClient Data Provider jako jednosměrný datový tok datového typu DBDataReader.

Vrstva EntityClient Data Provider provádí transformaci dat z datového modelu databáze na data konceptuálního modelu. Tato vrstva také provádí transformaci dotazů získaných z vrstvy Object Services na dotazy vyžadované vrstvou ADO.NET Data Providers. Klíčovou částí vrstvy EntityClient Data Provider je Entity Data Model (EDM, viz. 6.2.1). Vrstva EntityClient Data Provider je definována pomocí dvou modelů a mapování mezi nimi. První model mapuje skutečný relační datový model databáze v databázovém systému, druhý představuje konceptuální model dat, což je implementace EDM v technologii Entity Framework.

Jednotlivé modely a jejich vzájemné mapování je ve vrstvě EntityClient Data Provider definováno pomocí tři XML souborů: CSDL – definující konceptuální model, SSDL – definující datový model databáze, a MSL – definující mapování mezi konceptuálním a datovým modelem (Obrázek 4). Všechny tři soubory, pokud jsou generovány pomocí nástroje ADO.NET Entity Data Model Designer ze sady Visual Studio 2010, jsou uloženy v jednom souboru EDMX.

[5] command trees, jedná se o derivační stromy příkazů používané v technologii Entity Framework, jejich struktura může záviset na typu poskytovatele dat ADO.NET



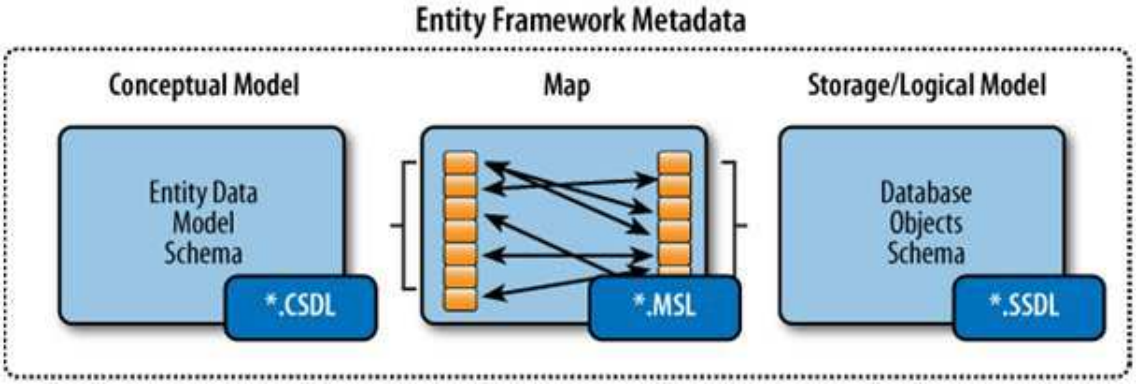
Obrázek 3: Architektura technologie Entity Framework [4]

Vrstva EntityClient Data Provider je dotazována přímo pomocí dotazů Entity SQL, nebo z vrstvy Object Services pomocí stromů příkazů. V obou případech vrací výsledná data jako datový tok EntityDataReader.

Nejvýše položenou vrstvou architektury Entity Framework je vrstva Object Services. Tato vrstva převádí data získaná z vrstvy EntityClient Data Provider na objekty – entity. Umožňuje s těmito entitami pracovat jako by to byla data z databáze, lze je načítat, upravovat, vkládat a mazat. Jádrem této vrstvy je třída ObjectContext (viz. 6.2.3). Entity lze z této vrstvy získat dotazováním, a to buď pomocí LINQ to Entities, nebo pomocí Entity SQL (viz. 6.3). Získané objekty jsou vráceny v kolekcích implementujících rozhraní IEnumerable<T>. [4]

6.2.1 Entity Data Model

Entity Data Model (EDM) je množina konceptů, která popisuje strukturu dat nezávisle na jejich uložené formě [5]. V technologii Entity Framework je implementován jako konceptuální model, který je definován pomocí XML dokumentu CSDL.



Obrázek 4: Jednotlivé modely a jejich XML dokumenty [12]

Struktura dat je v EDM reprezentována pomocí entit a vztahů mezi nimi. Vztahy mohou mít kardinalitu 1:1, 1:0..1, 1:n a dokonce i m:n. Pro použití vztahu m:n v EDM musí případná vazební tabulka v databázi obsahovat pouze klíče tabulek, jejichž vztah m:n určuje. Pokud jsou v této tabulce definovány i další atributy, je vytvořena vazební entita.

EDM definuje entitní typy, relační typy a komplexní typy. Jednotlivé instance entitních typů jsou definovány (unikátním) entitním klíčem, jednoznačným názvem, daty ve formě vlastností a navigačními vlastnostmi. Instance relačních typů představují vztahy mezi entitami a jsou definovány jednoznačným názvem, koncovými entitami a jejich klíči. Komplexní typy v EDM lze vkládat jako vlastnosti do jiných komplexních typů či do entitních typů a jejich instance jsou definovány jednoznačným názvem a daty ve formě vlastností.

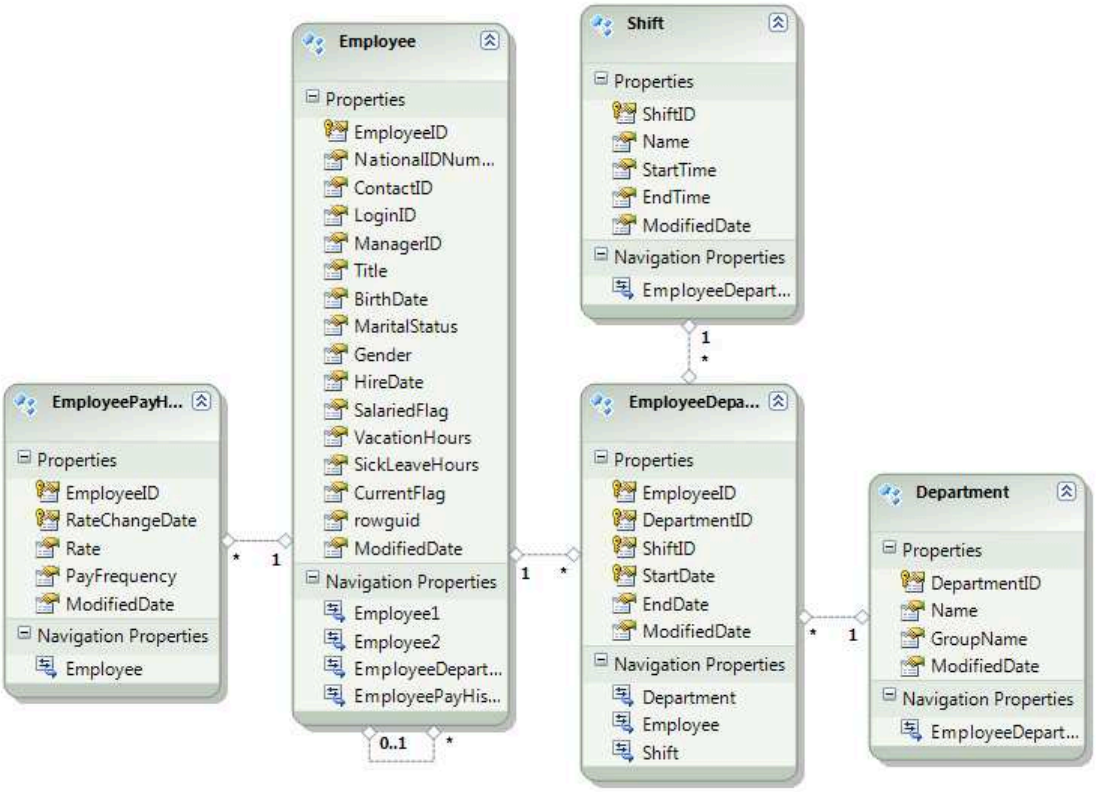
Následuje výčet a popis důležitých tříd vrstvy Object Services, která bude v této práci výhradně používána.

6.2.2 Třída EntityKey

Instance entitních typů (entity) používají k identifikaci entitní klíče, ty jsou reprezentovány instancí třídy EntityKey. Tato třída umožňuje identifikaci entity pomocí pole objektů EntityKeyMember, což jsou páry klíč-hodnota reprezentující název a hodnotu primárních atributů entity. Toto pole je uloženo ve vlastnosti EntityKeyValues. Třída EntityKey dále obsahuje atributy EntitySetName (název tabulky – množiny entit) a EntityContainerName (název kontejneru entit). Objekt EntityKey lze vytvořit samostatně, nebo pomocí metody CreateEntityKey konkrétní instance třídy ObjectContext. Objekt EntityKey je u objektů entit vytvářen automaticky.

6.2.3 Třída ObjectContext

Třída ObjectContext je jádrem vrstvy Object Services. Jedná se de facto o ekvivalent třídy DataContext ORM rámce LINQ to SQL. Třída ObjectContext zapouzdřuje objekty: En-



Obrázek 5: Schéma EDM v nástroji Entity Designer sady Visual Studio 2010

tityConnection (udržuje spojení s databází), MetadataWorkspace (představuje metadata popisující konceptuální model) a ObjectStateManager (hlídá objekty během vytváření, upravování a mazání).

6.2.4 Třída ObjectSet<TEntity>

Generická třída ObjectSet<TEntity> představuje silně typovanou kolekci entit, kde TEntity znamená entitní typ. Umožňuje vytvářet objekty zadaného entitního typu pomocí metody CreateObject(). Třída ObjectSet<TEntity> dědí ze třídy ObjectQuery<T>, umožňuje tedy provádět CRUD[6] operace. Obsahuje navíc například metody Attach() a Detach(). Pomocí metody Attach() lze do objektu ObjectSet<TEntity> přidávat objekty zadaného entitního typu, které byly například vytvořeny v jiném objektu ObjectContext. Metoda Detach() naopak umožňuje z objektu ObjectSet<TEntity> odebrat nepotřebné entitní objekty.

Generická třída ObjectSet<TEntity> rozšiřuje třídu ObjectQuery<T>, která disponuje metodou Include(). Ta umožňuje objektu ObjectSet<TEntity> definovat, které vlastnosti daného entitního objektu se mají plnit asociovanými entitními objekty.

[6]CReate Update Delete, standartní operace prováděné nad databází: vytvoření, změna a smazání

Objekt ObjectSet<TEntity> lze vytvořit metodou CreateObjectSet() třídy ObjectContext.

```
// Vytvoření objektu ObjectContext.
ObjectContext context =
    new ObjectContext("name=AdventureWorksEntities");

ObjectSet<Product> query = context.CreateObjectSet<Product>();
```

Výpis 7: Ukázka vytvoření objektu ObjectSet<Product> z objektu ObjectContext

6.3 Dotazování v technologii Entity Framework 4

Dotazování je aplikováno na konceptuální model a lze jej provádět buď přímo pomocí Entity SQL dotazů, kdy jsou výsledná data vrácená jako datový tok datového typu EntityDataReader, nebo přes vrstvu Object Services, která umožňuje dotazování pomocí Entity SQL i LINQ to Entities a získaná data převádí na kolekce entit.

6.3.1 Entity SQL

Jazyk Entity SQL vznikl jako primární dotazovací jazyk pro technologii Entity Framework. Ovšem s příchodem technologie LINQ začalo být preferováno dotazování pomocí LINQ dotazů. Syntaxe jazyka Entity SQL je podobná LINQ dotazům. Dotazy Entity SQL nejsou zapisovány přímo do kódu jako dotazy LINQ, ale jako řetězce typu string, tedy podobně jako SQL dotazy při použití technologie ADO.NET (Ukázka 8).

```
//z kontejneru context získá zaměstnance narozené před 8.8.1980
var employeesQuery = "SELECT␣VALUE␣e" +
                     "FROM␣epmloyeesObjectSet.Employee␣AS␣e" +
                     "WHERE␣e.BirthDate␣<␣'1980−08−08'";

//vytvoření objektů z dotazu
ObjectQuery<Employee> employeesObjects = context.CreateQuery<Employee>(employeesQuery);
```

Výpis 8: Ukázka Entity SQL dotazu

Dotazy Entity SQL lze rovněž skládat pomocí metod kolekcí určených k dotazování. Části Entity SQL dotazů jsou těmto metodám předávány jako řetězce (Ukázka 9).

```
//vytvoření dotazu pomocí metod kolekce (it je výchozí pojmenování pro proměnnou)
var employeesQuery = context.Employee
                     .Where("it.BirthDate␣<␣'1980−08−08'")
                     .OrderBy("it.BirthDate");
```

Výpis 9: Ukázka Entity SQL dotazu za použití metod

Dotazování pomocí Entity SQL lze s výhodou použít pro získání dat v datovém toku (použití s EntityClient vrstvou), nebo pokud je potřeba upravovat dotaz za běhu.

6.3.2 LINQ to Entites

Jedná se o implementaci technologie LINQ pro technologii Entity Framework. Dotazy technologie LINQ to Entities pracují pouze s objekty vrstvy Object Services, tedy s entitními třídami a objekty ObjectContext a ObjectSet<TEntity>. Více o technologii LINQ lze nalézt v kapitole 4.

6.3.2.1 Předkompilované dotazy

Jak bylo řečeno dříve, data jsou ze zdroje dat získávána až po enumeraci příslušné kolekce. Ve stejné době je i překládán dotaz technologie LINQ na svůj ekvivalent pro datový zdroj. Technologie LINQ to Entities a LINQ to SQL umožňují předkompilovat dotazy, které se používají vícekrát. Toto lze provést u dotazů bez parametrů i s proměnnými parametry (Ukázka 10).

```
//předkompilovaný dotaz pro vyhledání zaměstnanců narozených před zadaným datem
var compQuery = CompiledQuery.Compile<AdventureWorksEntities, DateTime, IQueryable<
    Employee>>
        ((AdventureWorksEntities ctx, DateTime birthday) =>
            from Employee e in ctx.Employee.OfType<Employee>()
            where e.BirthDate < birthday
            select e);
```

Výpis 10: Ukázka předkompilovaného dotazu

6.4 Práce s technologií Entity Framework 4

S technologii Entity Framework 4 lze pracovat jako s jakýmkoli jiným modulem technologie .NET Framework. Tedy napsat kód aplikace v libovolném textovém editoru (např. WordPad) a následně jej zkompilovat, případně použít vývojové prostředí (např. SharpDevelop). Vždy je samozřejmě nutné vytvořit tři XML soubory pro modely a mapování (s výjimkou přistupu Code-First, viz. 6.4.3) a importovat potřebné knihovny. Nejjednodušší cestou pro vytváření aplikací s použitím technologie Entity Framework 4 je vývoj v sadě Visual Studio 2010, která obsahuje nástroje pro automatické vytváření XML souborů vrstvy EntityClient.

Relační datový model lze v technologii Entity Framework 4 nasadit třemi způsoby: vytvořením v databázovém systému (Database First), vytvořením doménového modelu na doménové úrovni, a jeho následným promítnutím do databázového systému (Model First), nebo vytvořením datové struktury přímo v kódu aplikace (Code First).

6.4.1 Database-First

Implementace datového modelu přímo do databázového systému je obecně nejpoužívanější metodou pro vytváření relační datové struktury. Celá struktura datového modelu je vytvořena v databázovém systému a následně ji lze v technologii Entity Framework 4 namapovat pomocí SSDL souboru. V CSDL souboru lze dále datový model libovolně

upravovat podle potřeby dané aplikace, aniž by byl změněn datový model v databázovém systému. Mapování lze implementovat manuálně v souborech SSDL, MSL a CSDL, nebo pomocí nástrojů sady Visual Studio 2010, které tyto soubory vygenerují automaticky do jediného EDMX souboru.

6.4.2 Model-First

Entity Framework 4 umožňuje vytvářet relační datový model od doménové vrstvy. V sadě Visual Studio lze pomocí nástroje Entity Model Designer graficky vytvořit datový model aplikace jako doménový model. Z tohoto doménového modelu je následně vygenerován skript pro vytvoření databáze. Skript je vygenerován v SQL dialektu[7] specifickém pro požadovaný databázový systém (např.: T-SQL pro MS SQL Server). Při vytváření modelu je nutné mít v databázovém systému vytvořenou prázdnou databázi, jelikož Entity Framework 4 potřebuje při vytváření skriptu znát metadata popisující databázi v databázovém systému.

Tento přístup lze s výhodou použít, pokud při vývoji aplikace ještě nebyla v databázovém systému vytvořena databáze, nebo pokud vývojáři upřednostňují vytvoření nejprve konceptuálního datového modelu pro svou aplikaci.

6.4.3 Code-First

Přístup Code First umožňuje vývojářům pracovat s technologií Entity Framework aniž by museli používat XML soubory definující objektově relační mapování. Veškerá metadata mapování jsou vygenerována za běhu z kódu aplikace a následně uložena v paměti. Metadata jsou z kódu generována na základě používaných konvencí, např.: jako klíče jsou nastavovány vlastnosti obsahující v názvu ID. Tyto konvence lze obejít použitím anotací.

Mapování lze v tomto případě definovat pouze pomocí tzv. POCO tříd (viz. 6.5) a pomocí třídy definující, které POCO objekty mají vazbu na databázi, jedná se o tzv. kontextovou třídu. Tato třída musí dědit ze třídy DbContext a jednotlivé kolekce POCO objektů v ní musí být definovány jako vlastnosti typu DbSet<T>, což jsou generické kolekce vybraných POCO objektů. Databázi lze v tomto přístupu vytvořit dvěma způsoby: manuálně v databázovém systému, nebo automaticky přímo z výše definovaných tříd. Při automatickém generování databáze je po spuštění aplikace vyhledána databáze podle připojovacího řetězce se stejným názvem jako třída dědící ze třídy DbContext. V případě, že je potřeba výchozí nastavení mapování pomocí přístupu Code-First změnit, je nutné k tomu použít API.[11]

Přístup Code-First lze s výhodou použít, pokud není vyžadována přílišná konfigurace mapování. Nevýhodou mohou být malé možnosti automatizace generování kódu (zatím pouze pomocí vlastních T4 šablon[8]) a nutnost konfigurovat mapování pomocí API či anotací, pokud výchozí nastavení nevyhovuje. U větších informačních systému může být problematické uložení veškerého mapování v interní paměti.

[7] jazyk SQL uzpůsobený pro konkrétní databázové systémy

[8] šablona pro generování zdrojových kódu v technologii .NET Framework

Přístup Code-First lze v rámci technologie Entity Framework použít od verze 4.1. Následují některé další možnosti technologie Entity Framework.

6.4.4 Uložené procedury

Technologie Entity Framework 4 umožňuje taky práci s uloženými procedurami a funkcemi databáze. Tyto procedury mohou být namapovány na entitní typy, komplexní typy nebo na skalární typy (Ukázka 11).

V případě entitních typů se jedná především o mapování na operace insert, select a update, kdy jsou standardní SQL příkazy, které tyto operace běžně provádějí, nahrazeny právě procedurami. Po zavolání metody SaveChanges() objektu ObjectContext budou místo SQL příkazů spuštěny procedury. Toto mapování by mělo být provedeno u všech CRUD operací, jinak budou u všech těchto operací v některých případech opět prováděny SQL příkazy.

```
<Function Name="uspGetEmployeeManagers" Aggregate="false" BuiltIn="false" NiladicFunction="
    false" IsComposable="false" ParameterTypeSemantics="AllowImplicitConversion" Schema="
    dbo">
 <Parameter Name="EmployeeID" Type="int" Mode="In" />
</Function>
```

Výpis 11: Ukázka uložené procedury v souboru SSDL

6.4.5 Transakce

Veškeré transakce provádí objekt ObjectContext. Ten všechny akce nad databází, spojené s úpravami v paměti, provede po zavolání metody SaveChanges() jako jednu transakci. To znamená, že pokud se nepodaří provést minimálně jednu operaci nad databází, která byla spojena s úpravou, smazáním, nebo vytvořením objektu v paměti, povede to k akci rollback[9] v databázovém systému.

Technologie Entity Framework 4 umožňuje i použití vlastních transakcí. Ty lze s výhodou použít, například pokud pracujeme se dvěma a více objekty typu ObjectContext. Vlastní transakce lze definovat pomocí objektu TransactionScope ze jmenného prostoru System.Common (Ukázka 12). Tyto vlastní transakce lze použít i při práci s vrstvou EntityClient.

```
using (var transaction = new TransactionScope())
{
 firstContext.SaveChanges();

 var secondContext = new AdventureWorksEntities();
 secondContext.Employee.AddObject(new Employee { BirthDate = new DateTime(6, 6, 1980) });
                secondContext.SaveChanges();

 transaction.Complete();
 firstContext.AcceptAllChanges();
```

[9] vrácení všech změn od začátku transakce

```
        secondContext.AcceptAllChanges();
    }
```

Výpis 12: Ukázka použití objektu TransactionScope

6.5 Entity Framework 4 a POCO objekty

6.5.1 Co jsou to POCO objekty

POCO (Plain Old CLR Objects) objekty jsou objekty, které nejsou nijak závislé na rámcích, které s nimi pracují.[12] Tyto objekty dědí pouze ze třídy System.Object. V případě Entity Framework 4 se jedná o entity, které nedědí ze třídy EntityObject.

6.5.2 Proxy třídy

Entity Framework 4 umožňuje obalování POCO objektů instancemi tzv. proxy (zástupných) tříd. Ty zpřístupní POCO objektům možnosti, které mají entitní objekty technologie Entity Framework 4 dědící ze třídy EntityObject. Jedná se o možnosti: zpožděné vyhodnocování, notifikaci změn a opravu vazeb. Proxy objekty POCO objektům také přiřazují objekty typu EntityKey (viz. 6.2.2). Možnost zpožděného vyhodnocování je popsána v podkapitole 4.3. Možnost notifikace změn je důležitá pro zjišťování změn v objektech entit, v tomto případě POCO objektů. Na základě informací o změnách v objektech entit provede objekt ObjectContext příslušné změny v datovém zdroji. Možnost opravy vazeb mezi objekty entit zabraňuje tomu, aby vznikaly nekonzistence mezi cizími klíči a objekty v navigačních vlastnostech[10].

Aby byla technologie Entity Framework 4 schopná vytvořit pro POCO objekty proxy objekty, musí tyto objekty splňovat následující konvence. Všechny vlastnosti POCO tříd musí být označeny jako public a virtual, navigační vlastnosti nesmí být označeny jako sealed. POCO třídy nesmí být označeny jako sealed nebo abstrakt a musí obsahovat bezparametrický, případně žádný, konstruktor. Dále pak musí být všechny kolekce, obsahující asociované objekty, generického typu ICollection<T>. Aby byl POCO objekt obalen proxy objektem musí být navíc v příslušném objektu ObjectContext nastavena vlastnost ContextOptions.ProxyCreationEnabled na true a instance této POCO třídy musí být vytvořena pomocí metody CreateObject().

6.5.3 Proč POCO objekty

POCO objekty především umožňují snazší výměnu datové vrstvy, například pokud již máme nějaké existující třídy entit. Samostatnost POCO tříd umožňuje také snadnou binární serializaci a tedy jejich použití v SessionState, ViewState a podobných objektech technologie ADO.NET ukládajících data na pozadí webových stránek a také jako DTO (Data Transfer Object) pro přenos pomocí webových služeb či technologie WCF (Windows Communication Foundation). Lze provádět také serializaci obalujících proxy objektů. [6]

[10] v navigačních vlastnostech jsou uloženy odkazy na objekty, které jsou s objektem této vlastnosti v relaci

6.6 Porovnání s některými jinými ORM rámci

V následujícím textu jsou uvedeny podstatnější rozdíly mezi technologií Entity Framework 4 a některými dalšími ORM rámci.

6.6.1 Rozdíly mezi technologiemi Entity Framework 1 a Entity Framework 4

Entity Framework verze 4 oproti verzi 1:

- Přidává podporu pro POCO objekty
- Přidává podporu pro zpožděné vyhodnocování při získávání asociovaných objektů.
- Podporuje předávání entitních objektů mezi vrstvami n-vrstvých architektur společně se sledováním jejich stavu
- Vylepšuje generování SQL dotazů a podporu uložených procedur
- Vylepšuje testovatelnost
- Vylepšuje podporu operátorů technologie LINQ

[11]

6.6.2 Rozdíly mezi technologiemi Entity Framework a LINQ to SQL

Entity Framework má oproti LINQ to SQL možnost vytvoření doménové vrstvy pomocí EDM; může kromě databázového systému MS SQL verze 2000 a vyšší používat i jiné databázové systémy, pro které existuje poskytovatel dat pro ADO.NET; kromě technologie LINQ může pro dotazování používat jazyk Entity SQL. Technologie Entity Framework je obecně mnohem mohutnější nástroj, než technologie LINQ to SQL.

6.6.3 Rozdíly mezi technologiemi Entity Framework a NHibernate

ORM rámec NHibernate má za sebou oproti rámci Entity Framework daleko delší existenci a je tedy daleko „vyzrálejší“. Rámec NHibernate má více možností nastavení než Entity Framework, hlavně co se týká získávání záznamů z databáze a jejich převádění na entitní objekty. Tento rámec lze také rozšiřovat o další funkčnost. Oproti tomu Entity Framework má například (prozatím) lepší implementaci technologie LINQ, možnost vytvoření doménové vrstvy a jedná se hotové řešení přímo od společnosti Microsoft [9], které je navíc upřednostňovanou technologii pro objektově relační mapování v technologii .NET Framework. Společnost Microsoft navíc pro technologii Entity Framework vyvinula a podporuje nástroje pro automatizovanou tvorbu datové vrstvy integrované do sady Visual Studio.

7 Návrh a implementace datové vrstvy

Při návrhu datové vrstvy za použití návrhových vzoru Repository a Unit of Work jsem vycházel z T4 šablony vytvořené Danem Morgridgem[2] (verze z 2. června 2010). Kód této šablony je volně k dispozici podle licence: [3]. Z této šablony jsem použil především strukturu tříd implementujících návrhové vzory Repository a Unit of Work. K běžným metodám návrhového vzoru Repository jsem přidal metody pro přidávání a odebírání entit z objektu ObjectSet<TEntity> a metodu pro vytváření entit z tohoto objektu.

Návrhové vzory Repository a Unit of Work a POCO objekty jsou použity proto, aby byla usnadněna výměna datové vrstvy u již existujících aplikací. Níže popsaná implementace navíc umožňuje snadnou testovatelnost aplikací postavených nad touto datovou vrstvou.

7.1 Implementace POCO objektů

POCO objekty je nutné implementovat podle pravidel pro tvorbu POCO objektů, tak aby je technologie Entity Framework mohla obalit proxy objekty a zpřístupnila tak těmto třídám pokročilé možnosti technologie Entity Framework. Pravidla pro tvorbu POCO objektů jsou popsána v 6.5.2.

Pro POCO objekty je také nutné implementovat třídu dědící ze třídy ObjectContext, která bude umožňovat získávat objekty ObjectSet<TEntity> pro jednotlivé tabulky a definovat výchozí nastavení pro získávání dat pomocí objektu ObjectContext.

7.2 Implementace návrhového vzoru Unit of Work

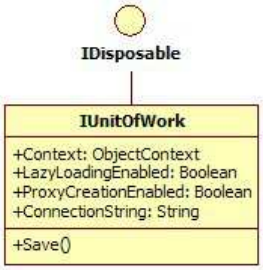
Implementace návrhového vzoru Unit of Work je provedena v rozhraní a v třídě, která toto rozhraní implementuje. Tato struktura usnadňuje díky rozhraní implementaci návrhového vzoru Unit of Work pro více objektově relačních rámců a umožňuje tedy snazší výměnu datové vrstvy. Návrhový vzor Unit of Work v této implementaci vlastně jen překrývá objekt ObjectContext a prezentuje jen některé jeho vlastnosti a funkce. Vzhledem k povaze a vztahu objektů ObjectContext a ObjectSet<TEntity> jsou metody pro vkládání a odebírání objektů implementovány návrhovým vzorem Repository. Metody registerDirty() a registerClean() nejsou implementovány, jelikož objekt ObjectContext již obsahuje shodnou funkčnost.

Následuje výčet a popis členů třídy EFUnitOfWork implementující rozhraní IUnitOfWork.

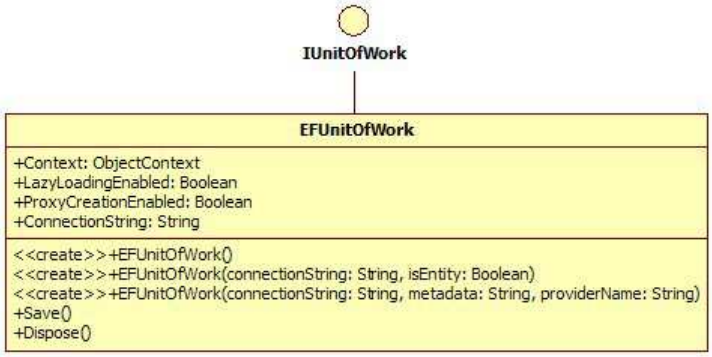
7.2.1 Konstruktory

Třída EFUnitOfWork má tři konstruktory: bezparametrický a dva se vstupními parametry. Bezparametrický konstruktor vytváří instanci třídy ObjectContext s výchozím nastavením řetězce připojení na databázi.

Konstruktor s parametry connectionString typu string a isEntity typu bool vytváří instanci třídy ObjectContext s řetězcem připojení na databázi z parametru connection-



Obrázek 6: Třídní diagram rozhraní IUnitOfWork



Obrázek 7: Třídní diagram třídy EFUnitOfWork

String. Parametr isEntity určuje, zdali je řetězec zadaný v parametru connectionString určený pro poskytovatele vrstvy EntityClient, nebo pro poskytovatele vrstvy ADO.NET. Jeĺi hodnota true jedná se o řetězec určený poskytovateli vrstvy EntityClient, v opačném případě je určený poskytovateli vrstvy ADO.NET. Pokud je řetězec určen pro poskytovatele vrstvy ADO.NET, je z tohoto řetězce, na základě výchozích parametrů tvorby EDM v nástroji pro tvorbu EDM v sadě Visual Studio 2010, sestaven řetězec pro poskytovatele vrstvy EntityClient.

Konstruktor s parametry connectionString typu string, metadata typu sting a providerName typu string také vytváří instanci třídy ObjectContext. Tento konstruktor ji vytváří s řetězcem připojení na databázi složeným z řetězců zadaných v parametrech. Parametr connectionString předává řetězec určený pro poskytovatele vrstvy ADO.NET, parametr metadata předává informace o XML souborech s definicemi jednotlivých vrstev EDM a parametr providerName obsahuje název poskytovatele dat vrstvy ADO.NET.

7.2.2 Vlastnosti

Třídě EFUnitOfWork dále obsahuje čtyři veřejné vlastnosti: vlastnost Context obsahující objekt typu ObjectContext a tři vlastnosti umožňující změnu některých vlastností tohoto objektu.

Vlastnost LazyLoadingEnabled typu bool určuje, zdali má přiřazený objekt typu ObjectContext mít schopnost zpožděného vyhodnocování. V případě, že je zpožděné vyhodnocování vypnuto, lze vybrané asociované objekty nahrát pomocí metody Include() objektu typu ObjectQuery. Metodě Include() je jako parametr typu string předáván řetězec obsahující cestu s názvem vlastnosti, která se má naplnit.

Vlastnost ProxyCreationEnabled typu bool určuje, zdali mají být POCO objekty vytvořené pomocí metody CreateObject obalený proxy objektem.

Vlastnost ConnectionString typu string umožňuje měnit řetězec připojení na databázi. Jedná se o řetězec pro poskytovatele vrstvy EntityClient.

7.2.3 Metoda Save()

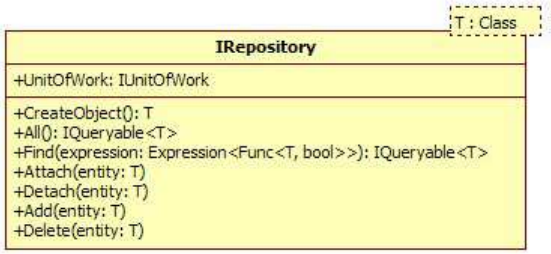
Metoda Save() je implementací metody commit() a promítá změny provedené v objektu typu ObjectContext, a objektech typu ObjectSet<TEntity> z tohoto objektu vytvořených, do databáze.

7.2.4 Metoda Dispose()

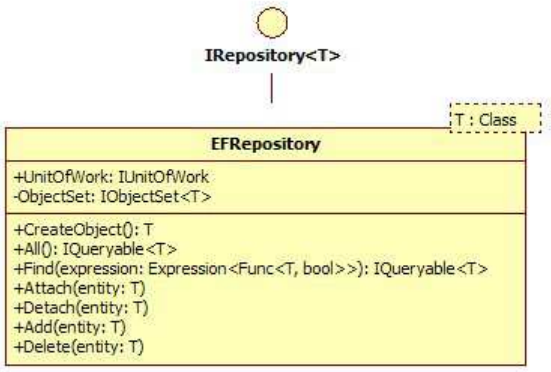
Implementace metody Dispose() rozhraní IDisposable v tomto objektu volá implementaci metody Dispose() přiřazeného objektu typu ObjectContext a ta zavírá připojení na databázi. Implementace rozhraní IDisposable umožňuje použití objektu v bloku using.

7.3 Implementace návrhového vzoru Repository

Návrhový vzor Repository je implementován jako generické rozhraní a generická třída toto rozhraní implementující. Stejně jako v případě návrhového vzoru Unit of Work umož-



Obrázek 8: Třídní diagram rozhraní IRepository



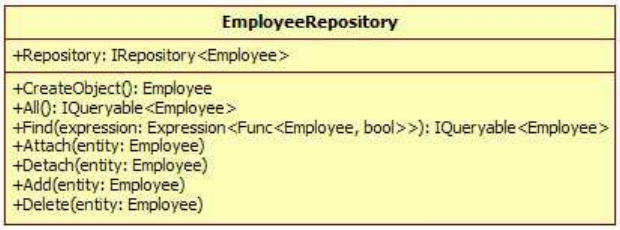
Obrázek 9: Třídní diagram generické třídy EFRepository

ňuje tato struktura díky rozhraní implementaci návrhového vzoru Repository pro více ORM rámců. Jako typ jsou pro tyto generické třídy použity vždy entitní typy, pro které jsou tyto třídy určeny. Dále jsou pro jednotlivé entity implementovány parciální třídy, které pracují s generickou třídou EFRepository<T> (kde T je entitní typ). Do těchto parciálních tříd lze vkládat kód pro získávání konkrétních dat z databáze. Parciální třída je použita proto, aby bylo možno co nejvíce kódu generovat pomocí T4 šablon.

Následuje výčet a popis členů třídy EFRepository<T>, která implementuje rozhraní IRepository<T> (kde t je entitní typ). Výčet metod parciálních tříd pro jednotlivé entitní typy je identický, liší se ve vlastnostech a konstruktoru.

7.3.1 Vlastnosti

Generická třída EFRepository<T> obsahuje public a private vlastnost. Protected vlastnost UnitOfWork obsahuje objekt typu IUnitOfWork asociovaný s aktuální instancí třídy. Private vlastnost ObjectSet obsahuje objekt typu ObjectSet<TEntity>, který byl vytvořen pro entitní typ T. Tato vlastnost má zpožděné vyhodnocování, při kterém je instance třídy



Obrázek 10: Třídní diagram třídy EmployeeRepository

ObjectSet<TEntity> vytvořena z vlastnosti Context objektu UnitOfWork pomocí metody CreateObjectSet<TEntity>().

Parciální třída má jedinou vlastnost: veřejnou vlastnost Repository typu IRepository<T>.

7.3.2 Konstruktor

Parciální třída obsahuje konstruktor se dvěma parametry. Jako první parametr je předávána instance třídy implementující rozhraní IUnitOfWork a jako druhý parametr instance třídy implementující rozhraní IRepository<T>. Právě toto umožňuje parciálním třídám předávat „falešné“ objekty implementující IUnitOfWork a IRepository<T>, které například nejsou připojené na databázi a testovat tak aplikace používající tuto datovou vrstvu.

7.3.3 Metoda CreateObject()

Metoda vytvoří a vrátí instanci entitního typu. Jelikož se v tomto případě jedná o POCO objekt, bude vrácena instance obalena proxy objektem (pokud je toto umožněno). I když je objekt vytvořen pomocí této metody, je nutné jej do objektu typu ObjectSet<TEntity> přidat pomocí metody Add().

7.3.4 Metoda All()

Metoda vrátí objekt implementující rozhraní IQueryable<T> s výběrem všech objektů reprezentující záznamy v tabulce.

7.3.5 Metoda Find()

Metoda vrátí objekt implementující rozhraní IQueryable<T> s výběrem všech objektů reprezentující záznamy v tabulce odpovídající podmínce zadané v parametru. Podmínka je typu Expression<Func<T, bool>>, takže je možno použít vlastní derivační strom.

7.3.6 Metoda Attach()

Metoda umožňuje vkládat do objektu uloženého ve vlastnosti ObjectSet objekty entitních typů z jiných objektů typu ObjectSet<TEntity>, které pracují se stejným entitním typem.

Objekty entit musí mít platný objekt typu EntityKey, pro POCO objekty to znamená, že musí být obaleny v proxy objektech. Pokud v datovém zdroji neexistují záznamy asociované s těmito objekty, budou tyto objekty zahozeny.

7.3.7 Metoda Detach()

Metoda umožňuje odebrat nepotřebný objekt z objektu uloženého ve vlastnosti ObjectSet. Tato změna není perzistentní, po zavolání metody Save() se se záznamem spojeným s objektem odstraněným tímto způsobem nic nestane. Odstraněn je vždy jen zvolený objekt a nikoli objekty s ním asociované.

7.3.8 Metoda Add()

Metoda označí objekt předaný v parametru jako přidaný do objektu uloženého ve vlastnosti ObjectSet.

7.3.9 Metoda Delete()

Metoda označí objekt předaný v parametru jako odstraněný z objektu uloženého ve vlastnosti ObjectSet.

7.4 Předpoklady pro vytvoření datové vrstvy v sadě Visual Studio 2010

Samozřejmostí pro práci se sadou Visual Studio 2010 a vytvoření datové vrstvy s POCO objekty by měla být technologie .NET Framework ve verzi, která zahrnuje technologii Entity Framework verze 4 a vyšší. Dále je vhodné mít nainstalovánu T4 šablonu pro tvorbu POCO tříd poskytovanou společností Microsoft. POCO třídy lze samozřejmě vytvořit také bez pomoci této šablony. Návrhové vzory Repository a Unit of Work lze naimplementovat podle slovního popisu a referenční implementace (Příloha A), použít upravenou šablonu [2], nebo si tuto šablonu upravit. Je samozřejmě také nutné vytvořit XML soubory popisující vrstvy EDM modelu. V případě, že byl EDM model vytvořen v grafickém nástroji sady Visual Studio 2010, je nutné nastavit vlastnost „Code Generation Strategy“ na hodnotu „none“.

Referenční implementaci návrhových vzorů pro část schématu HumanResources lze nalézt v příloze A, kde lze nalézt i její třídní diagram.

8 Práce s datovou vrstvou

V této kapitole jsou popsány základy práce s navrženou datovou vrstvou na vrstvě aplikační, případně klientské.

8.1 Vytváření instancí tříd EFUnitOfWork, EFRepository<T> a parciálních tříd Repository jednotlivých entitních typů

Pro práci s datovou vrstvou je vždy nutné vytvořit instanci třídy EFUnitOfWork, například v bloku using. Pro práci s entitami je pak potřeba vytvořit instanci příslušné Repository třídy. Této instanci je v konstruktoru předána instance třídy EFUnitOfWork a nová instance třídy EFRepository<T> (Ukázka 17). Tímto se instance Repository třídy připojí ke zvolenému objektu typu EFUnitOfWork. K objektu typu UnitOfWork může být připojena maximálně jedna instance Repository třídy konkrétního entitního typu.

8.2 Získávání entitních objektů z databáze

Entitní objekty, případně jejich kolekce, lze získat buď pomocí metod Find(), All() příslušného Repository (Ukázka 13), případně pomocí vlastních metod v tomto Repository implementovaných (Ukázka 14). Metodě Find() je v parametru předávána omezující podmínka typu Expression<Func<T, Bool>>, tuto podmínku lze tedy předat jako lambda výraz, anonymní třídu nebo jako strom příkazů. Metoda All() vrací všechny entitní objekty získané z objektu ObjectSet<TEntity>. Ve vlastních metodách lze pak definovat vlastní vyhledávání všemi možnostmi, které technologie Enntity Framework a LINQ nabízejí. Metody vracejí (případně měly by vracet) objekt jako rozhraní IQueryable<T> obsahující získané entitní objekty.

```
public void FindEmployeeBornBefore(DateTime birthDate)
{
    using (IUnitOfWork unitOfWork = new EFUnitOfWork())
    {
        EmployeeRepository employeeRepository =
            new EmployeeRepository(unitOfWork, new EFRepository<Employee>());

        Employee employee = employeeRepository.Find(e => e.BirthDate.CompareTo(birthDate) < 0).
            First();

        employeeRepository.Delete(employee);

        unitOfWork.Save();
    }
}
```

Výpis 13: Ukázka získání entity Employee z instance třídy EmployeeRepository

8.2.1 Spojování tabulek

Pro operace ekvivalentní se spojováním tabulek, případně pro jiné operace vyžadující pro svůj dotaz data z více tabulek, je nutné mít vytvořenu instanci tříd Repository příslušného entitního typu. Z nich poté použít metodu All(), s pomocí které lze získat přístup ke všem entitám daného entitního typu (Ukázka 14).

```
public partial class EmployeePayHistoryRepository
{
    public IQueryable<EmployeePayHistory> InnerJoin(DepartmentRepository
        departmentRepository,
        EmployeeDepartmentHistoryRepository employeeDepartmentHistoryRepository,
        EmployeeRepository employeeRepository, int departmentID)
    {
        var ret = from d in departmentRepository.All()
                  join edh in employeeDepartmentHistoryRepository.All() on d.DepartmentID equals
                      edh.DepartmentID into j1
                  from edh in j1
                  join e in employeeRepository.All() on edh.EmployeeID equals e.EmployeeID into j2
                  from e in j2
                  join eph in Repository.All() on e.EmployeeID equals eph.EmployeeID
                  where d.DepartmentID == departmentID
                  select eph;

        return ret.AsQueryable();
    }
}
```

Výpis 14: Ukázka získání kolekce objektů EmployeePayHistory jako IQueryable pomocí spojování tabulek

8.3 Vytváření nových entitních objektů

Jak bylo řečeno v 7.3.3 je vhodné vytvářet objekty pomocí metody CreateObject(). Po vytvoření objektu je nutné jej přidat do Repository pomocí metody Add() (Ukázka 15). Může se jednat i o Repository, které je spojeno s jiným objektem typu EFUnitOfWork.

```
public void NewEntity(DateTime birthDate)
{
    using (IUnitOfWork unitOfWork = new EFUnitOfWork())
    {
        EmployeeRepository employeeRepository =
            new EmployeeRepository(unitOfWork, new EFRepository<Employee>());

        Employee employee = employeeRepository.CreateObject();
        employee.BirthDate = birthDate;

        employeeRepository.Add(employee);

        unitOfWork.Save();
    }
}
```

Výpis 15: Ukázka vytvoření nové entity Employee a její následné vložení do objektu typu EmployeeRepository

8.4 Editace entitních objektů

Načtené či nově vytvořené entitní objekty lze upravovat jako jakékoli jiné objekty – pomocí jejich veřejně přístupných vlastností (Ukázka 15). Tyto úpravy jsou po zavolání metody Save() instance třídy EFUnitOfWork (ke které jsou tyto entitní objekty připojeny) uloženy do databáze.

8.5 Mazání entitních objektů

Objekty lze označit k vymazání pomocí metody Delete() (Ukázka 16). Záznam, na který byl tento objekt namapován, bude z datového zdroje odstraněn po zavolání metody Save() příslušného objektu typu UnitOfWork.

```
public void DeleteEntity(int  employeeID)
{
    using (IUnitOfWork unitOfWork = new EFUnitOfWork())
    {
        EmployeeRepository employeeRepository =
            new EmployeeRepository(unitOfWork, new EFRepository<Employee>());

        //metoda FindByID implementuje:
        //        return  Find(d => d.EmployeeID == id).Single();
        Employee employee = employeeRepository.FindByID(employeeID);

        employeeRepository.Delete(employee);

        unitOfWork.Save();
    }
}
```

Výpis 16: Ukázka označení entity Employee ke smazání

8.6 Připojování a odpojování entitních objektů

Připojování a odpojování existujících entitních objektů od objektu ObjectSet<T> lze provádět pomocí metod Attach() a Detach(). Tyto akce nemají vliv na vkládání a odebírání záznamů ve zdroji dat, jedná se pouze o programátorské přenášení načtených entitních objektů mezi objekty typu DataSet<T> a odstraňování nepotřebných entitních objektů z paměti.

```
public class ApplicationLayerClass
{
    EFUnitOfWork _unitOfWork;
```

```
    public ApplicationLayerClass()
    {
        _unitOfWork = new EFUnitOfWork();
    }

    public void AttachDetach()
    {
        EmployeeRepository employeeRepository1 =
            new EmployeeRepository(_unitOfWork, new EFRepository<Employee>());

        Employee employee = employeeRepository1.FindByID(100);

        employeeRepository1.Detach(employee);

        using (IUnitOfWork unitOfWork = new EFUnitOfWork())
        {
            EmployeeRepository employeeRepository2 =
                new EmployeeRepository(unitOfWork, new EFRepository<Employee>());

            employeeRepository2.Attach(employee);

            employee.BirthDate = new DateTime(1880, 10, 10);

            unitOfWork.Save();
        }
    }
}
```

Výpis 17: Ukázka odpojování a připojování entity Employee

8.7 Uložené procedury

Mapování procedur na skalární typy lze provádět v metodách třídy EFUnitOfWork. Příklad metody, která pracuje s uloženou procedurou lze nalézt v ukázce 18. Metoda v této ukázce předá proceduře parametry, spustí ji a vrátí počet řádků procedurou ovlivněných,

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;

namespace DataEF
{
    public partial class EFUnitOfWork
    {
        public int Procedure_uspUpdateEmployeeHireInfo(int employeeID, string title,
            DateTime hireDate, DateTime rateChangeDate, double rate, short payFrequency, int
                flag)
        {
            ObjectParameter[] parameters = new ObjectParameter[7];
```

```
                parameters[0] = new ObjectParameter("EmployeeID", employeeID);
                parameters[1] = new ObjectParameter("Title",  title ) ;
                parameters[2] = new ObjectParameter("HireDate", hireDate);
                parameters[3] = new ObjectParameter("RateChangeDate", rateChangeDate);
                parameters[4] = new ObjectParameter("Rate", rate);
                parameters[5] = new ObjectParameter("PayFrequency", payFrequency);
                parameters[6] = new ObjectParameter("Flag", flag);

                return (Context as AdventureWorksEntities)
                    .ExecuteFunction("uspUpdateEmployeeHireInfo", parameters);
            }
        }
    }
```

Výpis 18: Ukázka použití uložené procedury v partial třídě EFUnitOfWork

8.8 Předkompilované dotazy

Datová vrstva byla mimo jiné vytvořena s ohledem na co největší kontrolu nad objekty pomocí ní vytvořených. Proto je pro každou instanci generické třídy EFRepository<T> objekt typu ObjectSet<T> z objektu ObjectContext vytvořen pomocí metody CreateObjetSet<T>() a není načítán z jeho vlastností, do kterých není ani ukládán. Tento přístup ale v současné době neumožňuje vytváření předkompilovaných dotazů (viz. ukázka 10), které vyžadují objekt typu ObjectContext a objekty typu ObjectSet<T> z jeho vlastností.

9 Testování

V této kapitole budou rozebrány a slovně vyhodnoceny paměťové a rychlostní testy několika ORM rámců implementovaných v testovací aplikaci.

9.1 Výchozí podmínky testování

Ve všech testovacích případech byla u ORM rámců vypnuta funkčnost zpožděného vyhodnocování asociovaných objektů. U rámce NHibernate byla vlastnost dialect nastavrna na NHibernate.Dialect.MsSqlDialect2005, tzn. že byl použit SQL dialekt pro práci s databázovým serverem MS SQL 2005. Dále byl u rámce NHibernate použit pro dotazování jazyk HQL[11], u ostatní rámců byla pro dotazování použita technologie LINQ.

Jako zdroj dat byla použita ukázková databáze AdventureWorks pro databázový systém MS SQL Server 2005, konkrétně část jejího schématu HumanResources, nasazená na různých verzích databázového systému MS SQL Server. U testů jsem se zaměřil především na získávání entit z datového zdroje. Konkrétně se jedná o získání všech záznamů tabulky Employee, vyhledání a získání záznamu podle klíče z téže tabulky a získání záznamu z tabulky EmployeePayHistory pomocí dotazu spojujícího několik tabulek (Dotaz v technologii LINQ v ukázce 14). Testy byly prováděny na notebooku s následující konfigurací.

9.1.1 Konfigurace testovacího notebooku

- **Model:** Acer Aspire TimelineX
- **Procesor:** Intel Core i5 processor 430M (2,26 GHz, 3MB L3 Cache, 64 bit, 2 jádra, 4 vlákna)
- **Paměť:** 4 GB, DDR3, 1 GHz
- **Operační systém:** Windows 7 – Home Premium (64-bit)
- **HDD:** 640 GB, 5400 ot./min, 8 MB cache

Jednotlivé verze databázového systému MS SQL Server byly nasazeny ve virtuálním stroji, který byl spuštěn přímo na testovacím notebooku. Virtuální stroj byl hostován v aplikaci Oracle VM VirtualBox verze 4.0.4 r70112.

9.1.2 Konfigurace virtuálního stroje

- **Paměť:** 1 024 MB
- **Operační systém:** Windows 7 – Professional (32-bit)

[11] Hibernate Query Language, obdoba jazyka Entity SQL pro rámec Entity Framework

9.2 Paměťové testy

Paměťové testy byly prováděny nad databázovým systémem MS SQL Server 2008 R2. U POCO objektů byla vypnuta možnost vytváření proxy tříd. Měření bylo provedeno pomocí profilovacího software .NET Memory Profiler ve verzi 3.5 Professional.

9.2.1 Výsledky paměťových testů

V tabulkách je uveden počet objektů a sumy velikostí těchto objektů. Sloupec UOW obsahuje objekty typu Unit of Work (ObjectContext, ...). Sloupec Přístupová třída obsahuje objekty pracující s objekty typu Unit of Work (EFUnitOfWork, ...). Sloupec Employee reprezentuje objekty získané z tabulky Employee a sloupec EPH reprezentuje objekty získané z tabulky EmployeePayHistory. Objekty v těchto sloupcích představují získané entitní objekty, které zůstaly záměrně uchovány v paměti. Sloupec součet představuje součet všech vytvořených entitních objektů a celkovou velikost alokované paměti.

Samotný instanciovaný jmenný prostor NHibernate během paměťového testování vytvořil 8 187 objektů, které dohromady v paměti alokovaly 270 420 bajtů. Jmenný prostor System.Data.Entity vytvořil 14 602 objekty, které alokovaly 437 416 bajtů (při práci s rámcem Entity Framework 4 bez použití návrhových vzorů).

Počet obj./poč. bajtů	UOW	Přist. tř.	Employee	EPH	součet
Všechny záznamy	1 / 96	1 / 12	290 / 31 320	-	292 / 31 428
Záznam podle klíče	1 / 96	1 / 12	1 / 108	-	3 / 216
INNER JOIN	1 / 96	1 / 12	-	13 / 676	15 / 784

Tabulka 1: Výsledky paměťového testu ORM rámce Entity Framework 4 s použitím návrhových vzorů

Počet obj./poč. bajtů	UOW	Přist. tř.	Employee	EPH	součet
Všechny záznamy	1 / 96	1 / 24	290 / 32 480	-	292 / 32 600
Záznam podle klíče	1 / 96	1 / 24	1 / 112	-	3 / 232
INNER JOIN	1 / 96	1 / 24	-	13 / 884	15 / 1 004

Tabulka 2: Výsledky paměťového testu ORM rámce Entity Framework 4

Počet obj./poč. bajtů	UOW	Přist. tř.	Employee	EPH	součet
Všechny záznamy	1 / 112	1 / 12	290 / 31 320	290 / 15 080	897 / 64 925
Záznam podle klíče	1 / 112	1 / 12	234 / 25 272	234 / 12 168	722 / 52 384
INNER JOIN	1 / 112	1 / 12	234 / 25 272	234 / 12 168	722 / 52 384

Tabulka 3: Výsledky paměťového testu ORM rámce NHibernate

Při získávání všech možných objektů z tabulky Employee vytvořil ORM rámec NHibernate, mimo objekty uvedené v tabulce, 296 objektů typu EmployeeDepartmentHis-

tory o celkové velikosti 17 760 bajtů, 16 objektů typu Department o celkové velikosti 512 bajtů a 3 objekty typu Shift o celkové velikosti 132 bajtů.

Při získávání objektu podle klíče i pomocí spojování tabulek vytvořil ORM rámec NHibernate, mimo objekty uvedené v tabulce, 240 objektů typu EmployeeDepartmentHistory o celkové velikosti 14 400 bajtů, 9 objektů typu Department o celkové velikosti 288 bajtů a 3 objekty typu Shift o celkové velikosti 132 bajtů.

9.2.2 Vyhodnocení paměťových testů

Z výsledků testů vyplývá, že nejmenší entitní objekty vytváří rámce NHibernate a Entity Framework 4 s použitím návrhových vzorů (108 bajtů pro entitní typ Employee). Velikost je pro oba rámce shodná jelikož používali podobné POCO objekty. Pokud jsou ovšem POCO objekty rámce Entity Framework 4 s použitím návrhových vzorů obaleny proxy objekty, je výsledná velikost entitních objektů podstatně větší (120 bajtů pro entitní typ Employee). Velikost entitních objektů rámce Entity Framework 4 dědících ze třídy Entity leží na pomyslné ose mezi velikostí POCO objektů a POCO objektů obalených proxy objekty (112 bajtů pro entitní typ Employee).

Pokud budeme o třídě EFUnitOfWork uvažovat jako o přístupovém objektu k objektu ObjectContext, lze říci, že jeho použití s objekty typu Repository je paměťově výhodnější (velikost objektu typu EFUnitOfWork byla 12 bajtů), než použití jednoduchých tříd obsahujících všechny metody a objekty požadované pro přístup k datovému zdroji (velikost objektu typu EFSimpleAccess byla 24 bajtů). Je nutno vzít v potaz, že třída EFSimpleAccess navíc obsahovala tři předkompilované dotazy.

V případě rámce NHibernate došlo k nežádoucímu nárůstu počtu entitních objektu v paměti, přestože bylo vypnuto zpožděné vyhodnocování. Na druhou stranu jmenný prostor System.Data.Entity, který je nutný pro běh ORM rámce Entity Framework 4, alokoval podstatně více paměti (437 416 bajtů), než knihovny ORM rámce NHibernate (270 420 bajtů).

9.3 Rychlostní testy

U rámců byla testována rychlost získání objektů od zaslání dotazu. Objekty byly získávány z předem inicializovaných objektů typu Unit of Work a objektů s nimi manipulujících, případně byly tyto objekty vytvořeny pouze pro dotaz a následně zahozeny (v tomto případě byla měřena i délka jejich inicializace). U ORM rámce Entity Framework 4 bez použití návrhových vzorů byla navíc testována rychlost při předkompilování LINQ dotazů.

9.3.1 Výsledky rychlostních testů

Každý test byl proveden 200 krát v sérii a výsledné časy byly sečteny a zprůměrovány. Při testech nebylo počítáno s časy prvního spuštění. Doba vyhodnocování byla měřena v milisekundách a to na vyšší vrstvě testovací aplikace jako doba mezi zavoláním me-

tody datové vrstvy, případně navíc vytvořením objektu typu Unit of Work, a vrácením zpracovaných entitních objektů.

V následujících tabulkách jsou uvedeny změřené časy pro jednotlivé verze databázového systému MS SQL Server. Názvy sloupců se vztahují k verzi tohoto databázového systému.

- Výsledky pro ORM rámec Entity Framework 4 s použitím návrhových vzorů

doba v ms	2005	2008	2008 R2
Všechny záznamy	11	5	4
Záznam podle klíče	5	5	4
INNER JOIN	20	19	18

Tabulka 4: Výsledky rychlostních testů ORM rámce Entity Framework 4 s použitím návrhových vzorů (objekt typu Unit of Work a objekt s ním manipulujícím byl vytvořen před testy)

doba v ms	2005	2008	2008 R2
Všechny záznamy	22	20	18
Záznam podle klíče	5	6	5
INNER JOIN	21	20	20

Tabulka 5: Výsledky rychlostních testů ORM rámce Entity Framework 4 s použitím návrhových vzorů (objekt typu Unit of Work a objekt s ním manipulující byl vytvářen během testů)

- Výsledky pro ORM rámec Entity Framework 4

doba v ms	2005	2008	2008 R2
Všechny záznamy	10	5	4
Záznam podle klíče	5	5	4
INNER JOIN	19	19	18

Tabulka 6: Výsledky rychlostních testů ORM rámce Entity Framework 4 (objekt typu Unit of Work a objekt s ním manipulujícím byl vytvořen před testy, dotazy nebyly předkompilovány)

doba v ms	2005	2008	2008 R2
Všechny záznamy	20	14	13
Záznam podle klíče	5	6	5
INNER JOIN	22	21	20

Tabulka 7: Výsledky rychlostních testů ORM rámce Entity Framework 4 (objekt typu Unit of Work a objekt s ním manipulující byl vytvářen během testů, dotazy nebyly překompilovány)

doba v ms	2005	2008	2008 R2
Všechny záznamy	7	5	4
Záznam podle klíče	1	1	1
INNER JOIN	1	1	1

Tabulka 8: Výsledky rychlostních testů ORM rámce Entity Framework 4 (objekt typu Unit of Work a objekt s ním manipulujícím byl vytvořen před testy, dotazy byly předkompilovány)

doba v ms	2005	2008	2008 R2
Všechny záznamy	25	17	16
Záznam podle klíče	6	5	5
INNER JOIN	22	21	20

Tabulka 9: Výsledky rychlostních testů ORM rámce Entity Framework 4 (objekt typu Unit of Work a objekt s ním manipulující byl vytvářen během testů, dotazy byly překompilovávány během testů)

- Výsledky pro ORM rámec NHibernate

doba v ms	2005	2008	2008 R2
Všechny záznamy	4	5	4
Záznam podle klíče	1	2	1
INNER JOIN	1	2	1

Tabulka 10: Výsledky rychlostních testů ORM rámce NHibernate (objekt typu Unit of Work a objekt s ním manipulující byl vytvořen před testy)

doba v ms	2005	2008	2008 R2
Všechny záznamy	1 287	1 780	1 321
Záznam podle klíče	1 020	1 407	1 051
INNER JOIN	1 025	1 418	1 056

Tabulka 11: Výsledky rychlostních testů ORM rámce NHibernate (objekt typu Unit of Work a objekt s ním manipulující byl vytvářen během testů)

9.3.2 Vyhodnocení rychlostních testů

Časové rozdíly při vytváření připojení a vyhodnocování dotazů mezi databázovými systémy MS SQL Server verze 2008 a 2008 R2 lze vysvětlit optimalizací tohoto databázového systému. V případě ORM rámce Entity Framework 4 lze rozdíl v době vyhodnocování mezi verzí 2005 a verzemi 2008 a 2008 R2 vysvětlit jeho lepší optimalizací právě pro novější verze tohoto systému. U ORM rámce NHibernate lze pozorovat prodloužení doby vyhodnocování mezi verzí 2005 a verzí 2008 a zkrácení doby vyhodnocování mezi verzí 2008 a 2008 R2. To lze ,v případě verzí 2005 a 2008, vysvětlit použitím překladače HQL dotazů pro SQL dialekt verze 2005 a, v případě verzí 2008 a 2008 R2, optimalizací databázového systému.

Celkově nejrychleji vyhodnocoval rámec Entity Framework 4 bez použití návrhových vzorů a s použitím předkompilování dotazů technologie LINQ. Dále nejrychleji vyhodnocoval rámec NHibernate, ale pouze pokud byl objekt typu Unit of Work vytvořen předem. Bez použití předkompilování dotazů technologie LINQ vyhodnocoval ORM rámec Entity Framework bez i s návrhovými vzory srovnatelně stejně dlouhou dobu. Nesrovnatelně nejdéle vytvářel objekt typu Unit of Work rámec NHibernate.

10 Závěr

Technologie Entity Framework 4 je mocným nástrojem pro tvorbu datových vrstev. Především díky možnosti vytvoření doménové vrstvy a zároveň rychlému vyhodnocování dotazů hlavně v případě jejich předkompilování. Tato technologie je snadno a rychle použitelná díky nástrojům, které pro ni byly vytvořeny. Její nevýhodou mohou být například omezené možnosti jejího rozšíření o vlastní funkčnost.

Navržená datová vrstva, která implementuje vrstvu návrhových vzorů nad technologií Entity Framework 4, umožňuje snadnou testovatelnost funkčnosti aplikací nad ní postavených a snadnou výměnu datové vrstvy. Toto je však vykoupeno nemožnosti použití předkompilování dotazů a tedy menší rychlostí vyhodnocování dotazů. Výhodou navržené datové vrstvy mohou být například POCO objekty, které jsou kompaktní a snadno serializovatelné.

Rozhodnutí, jestli použít tuto datovou vrstvu, nebo jiné řešení, závisí především na prioritách vývoje, testování a následném nasazení výsledného produktu.

11 Reference

[1] LINQ: .NET Language-Integrated Query. [online], únor 2007 [cit. 2010-5-12].
Dostupné z WWW: <http://msdn.microsoft.com/cs-cz/library/bb308959%28en-us%29.aspx>

[2] Entity Framework Repository & Unit of Work Template. [online], 2. červen 2010 [cit. 2010-06-10].
Dostupné z WWW: <http://efrepository.codeplex.com/>

[3] Entity Framework Repository & Unit of Work Template - license. [online], 2. červen 2010 [cit. 2010-06-10].
Dostupné z WWW: <http://efrepository.codeplex.com/license>

[4] ADO.NET Entity Framework. [online], c2010 [cit. 2010-10-12].
Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/bb399572.aspx>

[5] Entity Data Model. [online], c2010 [cit. 2010-11-20].
Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ee382825.aspx>

[6] Working with POCO Entities. [online], c2010 [cit. 2011-2-4].
Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/dd456853.aspx>

[7] MSDN Library. c2011 [cit. 2011-04-20].
Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ms123401.aspx>

[8] Alexander, C.: *A Pattern Language*. Oxford, 1977.

[9] Eini, O.: NHibernate vs. Entity Framework 4.0. [online], 5. leden 2010 [cit. 2011-4-20].
Dostupné z WWW: <http://ayende.com/Blog/archive/2010/01/05/nhibernate-vs.-entity-framework-4.0.aspx>

[10] Erich Gamma, R. J. J. V., Richard Helm: *Design Patterns: Elements of Reusable Object Oriented Software*. Addison Wesley, 1995.

[11] Guthrie, S.: Code-First Development with Entity Framework 4. [online], 16. červenec 2010 [cit. 2011-03-10].
Dostupné z WWW: <http://weblogs.asp.net/scottgu/archive/2010/07/16/code-first-development-with-entity-framework-4.aspx>

[12] Lerman, J.: *Programming Entity Framework, 2nd Edition*. O'Reilly, 2010.

[13] Mains, B.: Introduction to 3-Tier Architecture. [online], 28. duben 2008 [cit. 2011-5-1]. Dostupné z WWW: <http://dotnetslackers.com/articles/net/IntroductionTo3TierArchitecture.aspx>

[14] Martin Fowler, M. F. E. H. R. M. R. S., David Rice: *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002.

[15] Pichlík, R.: Třívrstvá architektura v kostce I. [online], 11. listopad 2004 [cit. 2010-02-04].
Dostupné z WWW: <http://www.dagblog.cz/2004_11_07_archive.html>

[16] Stanek, I.: *Objektově relační mapování pro platformu .NET*. Diplomová práce, České vysoké učení technické v Praze, 2008.

A Referenční implementace návrhových vzorů

V této příloze je uveden kód referenční implementace návrhových vzorů Repository a Unit of Work za použití technologie Entity Framework 4. Jako zdroj dat byla použita část schématu HumanResources ukázkové databáze AdventureWorks.

A.1 Návrhový vzor Unit of Work

```
using System.Data.Objects;
using System;

namespace DataEF
{
    public interface IUnitOfWork : IDisposable
    {
        ObjectContext Context { get; set; }
        void Save();
        bool LazyLoadingEnabled { get; set; }
        bool ProxyCreationEnabled { get; set; }
        string ConnectionString { get; }
    }
}
```

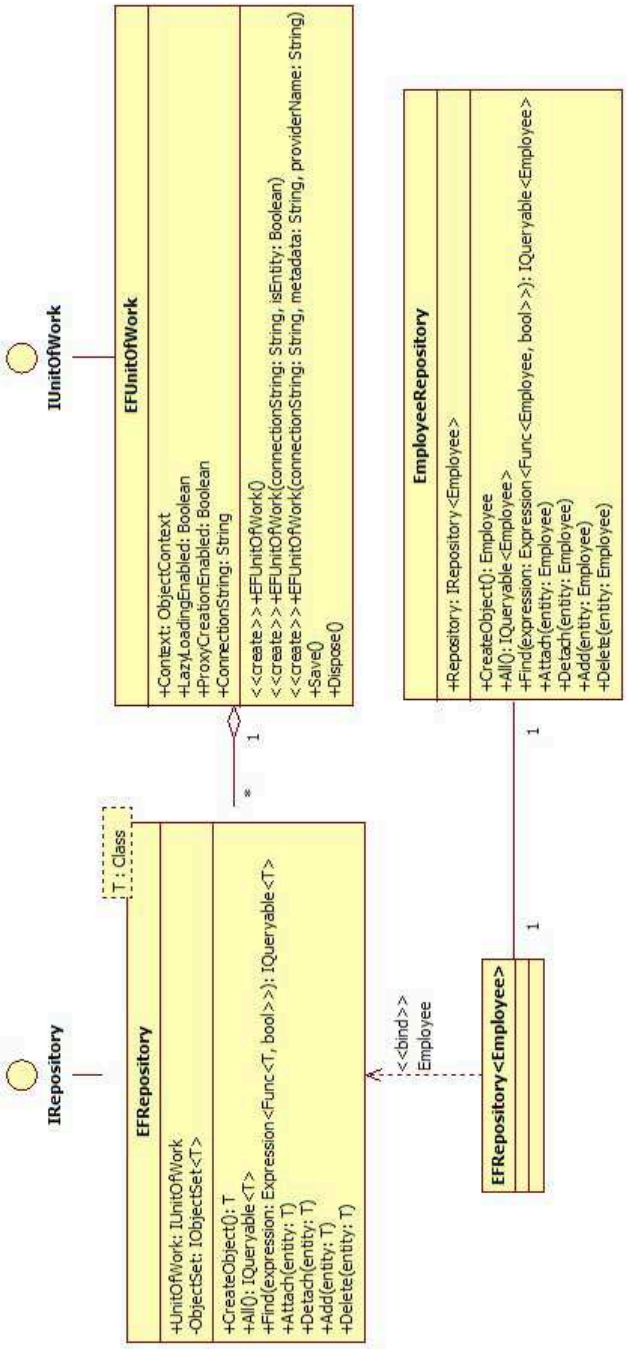
Výpis 19: Rozhraní IUnitOfWork

```
using System;
using System.Data.Objects;
using System.Data.EntityClient;
using System.Data.SqlClient;

namespace DataEF
{
    public class EFUnitOfWork : IUnitOfWork
    {
        public ObjectContext Context { get; set; }

        public EFUnitOfWork()
        {
            Context = new AdventureWorksEntities();
        }

          public EFUnitOfWork(string connectionString, bool isEntity)
          {
            if (! isEntity )
            {
                    EntityConnectionStringBuilder ecsb = new EntityConnectionStringBuilder
                    {
                        ProviderConnectionString = connectionString,
                        Provider = "System.Data.SqlClient",
                        Metadata = "res ://*/ AdventureWorksModel.csdl|res://*/AdventureWorksModel.
                            ssdl|res://*/AdventureWorksModel.msl"
                    };
```



Obrázek 11: Třídní diagram referenční implementace

```
                connectionString = ecsb.ToString();
            }

            Context = new AdventureWorksEntities(connectionString);
        }

        public EFUnitOfWork(string connectionString, string metadata, string providerName)
        {
            EntityConnectionStringBuilder ecsb = new EntityConnectionStringBuilder
            {
                ProviderConnectionString = connectionString,
                Provider = String.IsNullOrEmpty(providerName) ? "System.Data.SqlClient" :
                    providerName,
                Metadata =
                String.IsNullOrEmpty(metadata) ?
                    "res://*/ AdventureWorksModel.csdl|res://*/AdventureWorksModel.ssdl|res://*/
                        AdventureWorksModel.msl" :
                    metadata
            };

            Context = new AdventureWorksEntities(ecsb.ToString());
        }

    public bool LazyLoadingEnabled
    {
        get { return Context.ContextOptions.LazyLoadingEnabled; }
        set { Context.ContextOptions.LazyLoadingEnabled = value;}
    }

    public bool ProxyCreationEnabled
    {
        get { return Context.ContextOptions.ProxyCreationEnabled; }
        set { Context.ContextOptions.ProxyCreationEnabled = value; }
    }

    public string ConnectionString
    {
        get { return Context.Connection.ConnectionString; }
        set { Context.Connection.ConnectionString = value; }
    }

    public void Save()
    {
        Context.SaveChanges();
    }

    public void Dispose()
    {
        Context.Dispose();
    }
  }
}
```

Výpis 20: Třída EFUnitOfWork

A.2 Návrhový vzor Repository

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Linq.Expressions;

namespace DataEF
{
    public interface IRepository<T>
     {
        IUnitOfWork UnitOfWork { get; set; }
        T CreateObject();
        IQueryable<T> All();
        IQueryable<T> Find(Expression<Func<T, bool>> expression);
        void Attach(T entity);
        void Detach(T entity);
        void Add(T entity);
        void Delete(T entity);
     }
}
```

Výpis 21: Rozhraní IRepository

```
using System;
using System.Collections.Generic;
using System.Data.Objects;
using System.Linq;
using System.Linq.Expressions;
using System.Text;

namespace DataEF
{
    public class EFRepository<T> : IRepository<T> where T : class
    {
        protected IUnitOfWork UnitOfWork { get; set; }
        private IObjectSet<T> _objectset;
        private IObjectSet<T> ObjectSet
        {
            get
            {
                if (_objectset == null)
                {
                    _objectset = UnitOfWork.Context.CreateObjectSet<T>();
                }
                return _objectset;
            }
```

```
            set
            {
                _objectset = value;
            }

        }

        public T CreateObject()
        {
            return UnitOfWork.Context.CreateObject<T>();
        }

        public virtual IQueryable<T> All()
        {
            return ObjectSet.AsQueryable();
        }

        public IQueryable<T> Find(Expression<Func<T, bool>> expression)
        {
            return ObjectSet.Where(expression);
        }

        public void Attach(T entity)
        {
            ObjectSet.Attach(entity);
        }

        public void Detach(T entity)
        {
            ObjectSet.Detach(entity);
        }

        public void Add(T entity)
        {
            ObjectSet.AddObject(entity);
        }

        public void Delete(T entity)
        {
            ObjectSet.DeleteObject(entity);
        }
    }
}
```

Výpis 22: Třída EFRepository

V ukázce 23 je uvedena implementace parciální třídy pro entitní typ Employee ze schématu HumanResources ukázkové databáze AdventureWorks.

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.Linq.Expressions;
```

```
namespace DataEF
{
    public partial class EmployeeRepository
    {
        public IRepository<Employee> Repository { get; set; }

        public EmployeeRepository(IUnitOfWork unitOfWork, IRepository<Employee> repository)
        {
            Repository = repository;
            Repository.UnitOfWork = unitOfWork;
        }

        public Employee CreateObject()
        {
            return Repository.CreateObject();
        }

        public IQueryable<Employee> All()
        {
            return Repository.All();
        }

        public IQueryable<Employee> Find(Expression<Func<Employee, bool>> expression)
        {
            return Repository.Find(expression);
        }

        public void Attach(Employee entity)
        {
            Repository.Attach(entity);
        }

        public void Detach(Employee entity)
        {
            Repository.Detach(entity);
        }

        public void Add(Employee entity)
        {
            Repository.Add(entity);
        }

        public void Delete(Employee entity)
        {
            Repository.Delete(entity);
        }
    }
}
```

Výpis 23: Třída EmployeeRepository

B Obsah CD

- DPTestApplication_src - složka obsahující zdrojové kódy testovací aplikace
- DPTestApplication - složka obsahující spustitelný soubor testovací aplikace, kompilované knihovny a další zdroje potřebné pro běh testovací aplikace
- BAS064_dipl.pdf - tento text ve formátu PDF/A
- prilohy.pdf - přílohy diplomové práce ve formátu PDF/A
- zadani_sign.pdf - naskenované zadání diplomové práce ve formátu PDF/A